Appendix 1. New Input Formats and Utilities

A set of new input utilities were developed for MF-OWHM2 and are used by most of the new features presented in this report. The new utilities simplify input by the use of keywords and generalize the input structure of input and output (I/O) files. This new structure simplifies the name file (NAM)—the file that specifies the names of all global I/O data files and packages used in the model simulation and controls the parts of the model program that are active—and allows for an easier understanding of the MF-OWHM2 input for someone who did not build the original simulation model. Further, it allows for a standardized input structure that is easier to develop and maintain for the original model developer. These tools are currently only fully supported in the Farm Process version 4 (FMP) and select packages. With each subsequent release of MF-OWHM2, the utilities can be folded more into the traditional MODFLOW packages to simplify their input.

This simplification is done with keywords that indicate the frequency with which the input is loaded (**TEMPORAL_KEY**), the input style that is loaded (**INPUT_STYLE**), where the input is located (*Generic_Input*), and a host of options for applying multiple scale factors (for example, to separate out conversion factors, transformation multipliers, and calibration parameters/multipliers). Each of the input utilities builds upon each other with the simplest being the *Generic_Input. Generic_Input* is used by the *Universal Loader* (ULOAD) to read input, which is then called by the *Transient File Reader* (TFR) for spatially and temporally varying input, which is then called by the *List-Array Input* utility that parses the input style and temporal frequency. If the data are loaded only once and used for the entire simulation, then the *List-Array Input* utility bypasses the *Transient File Reader* and directly uses ULOAD one time. The temporally varying input can be read at any time interval (for example, time step or stress period) defined by the Package that uses the utilities. The rest of this appendix uses the term "stress period" synonymously with "time interval" because the stress period is the fundamental time interval used by MODFLOW to receive and hold constant user-specified inflows and outflows.

This appendix begins by describing the use of comments in packages and describing a new "*Block Style*" input format. The rest of the appendix begins with a top-down flow chart of the *List-Array Input* utility's structure. This is to illustrate the effect each keyword has until the final input is loaded. The next section discusses the seven possible combinations of keywords available with the *List-Array Input* and briefly discusses them. The flow chart provides a road map of the subsequent sections that summarize the major input utilities developed for MF-OWHM2. Each of the utilities makes use of the previous ones in this appendix. The discussion of each utility includes a top-down overview flow chart of the keywords that make up the *List-Array Input*, a bottom up description of the input utilities for the *Generic_Input* and *Generic_Output*, the ULOAD and scale factors (defined by the keyword **SFAC**), and finally the *List-Array Input* Style and *Transient File Reader*. The appendix concludes with a formal description of the *IXJ Style* input and the *Lookup Style* input.

Comments in Package Input

A new basic read utility, *READ_TO_DATA*, was developed to allow comments in input files. It has been applied entirely to the Farm Process (FMP), General Head Boundary (GHB), and Well (WEL) packages and incorporated at select locations in other packages. A commented line contains the symbol, #, with only blank space to the left of it. Comments to the right of a line with input values should also be preceded by a "#" symbol. This ensures that the comment is not treated as an input value. Empty lines, although not a comment, are automatically skipped. Examples of commented lines are shown in figure 1.1.

Block-Style Input

MODFLOW 2005 (Harbaugh, 2005) input relied on positional integer flags (for example, 2, 4, 6) and floating-point numbers (for example, 2.0, 4.2, 6.E8) to construct a simulation model. The new block-style input was intended to increase user friendliness, ease of use and documentation, and flexibility. A block input nests model-package input properties in one location (both temporally varying and static inputs), providing a simple input structure that helps the user understand what is being supplied to a model. Each block begins with the word **BEGIN** followed by the block's **NAME**, which defines the input in the block, then the block terminates its input feed with the word **END**. To the right of the block name is an optional set of **GLOBAL_KEYWORD**s that alter the behavior of the entire block. In the block is its input, which relies on keywords to define each input type and its style (for example, transient input or static input). Block input allows keywords to be in any order; it automatically skips blank lines; and comments are accepted as long as they are preceded with a # symbol. Figure 1.2 presents the general structure of an input block.

The block specific keyword **BLOCK_INCLUDE** mimics the functionality of the C language `#include "file"` and Fortran `INCLUDE File` statements allowing the user to specify a part of a block in a separate file that is inserted at the **BLOCK_INCLUDE** location. This feature is useful if a block needs to be subdivided into individual files or to use the separate file as a calibration template file. Figure 1.3 is a simple example of **BLOCK_INCLUDE** that loads four keywords. At runtime, before any keywords are processed, the block input inserts all **BLOCK_INCLUDE** files, strips from the block any comments and blank lines, and adjusts all remaining lines in the block to be left justified (fig. 1.3*C*).

```
             Column Number                                                          Column Number
 1234567890123456789012345678                                                       12345678901234
|55  65       # Side comment |  →  MF-OWHM2 only sees  "55  65        "
|   55  65    # Side comment |  →  MF-OWHM2 only sees  "   55  65     "
|#  55  65    # Side comment |  →  MF-OWHM2 skips the line and loads the next.
|  #55  65    # Side comment |  →  MF-OWHM2 skips the line and loads the next.
|                            |  →  MF-OWHM2 skips empty line and loads the next.
```

Figure 1.1. Example input lines, delineated with a |, that are parsed by MF-OWHM2 to remove comments and blank lines. To the right of each example input line is an explanation of what MF-OWHM sees as input. [Note that a "|" is used to delineate the start and end of each example input line. Comments must be preceded by a # (pound sign). Column number is the number of spaces within the file ranging from 1 to 28.]

A

```
BEGIN NAME    GLOBAL_KEYWORD
              #
              # Comments and blank lines are ignored

              #
              KEYWORD      # COMMENT

              # COMMENT
              KEYWORD      # COMMENT
              # COMMENT
              #
              BLOCK_INCLUDE  Generic_Input_OptKey
              #
END
```

Figure 1.2. Block-Style Input *A*, structure, and *B*, explanation of the items in part *A*.

B

BEGIN	initiates the block input.
NAME	is the block input name that identifies the expected input within the block.
#	the # and anything to the right of the # is ignored.
KEYWORD	is a keyword that is defined for use within the block.
END	must be present and terminates reading of the block input. Text after **END** is ignored, so it is recommended to add the block's name after the termination word for clarity (for example, **END NAME**).
GLOBAL_KEYWORDS	is optional; if present, a set of global keywords that affect the entire block. An example is **BEGIN LINEFEED XY**, where the block name, **LINEFEED**, indicates that the block contains LineFeed input and the global block keyword, **XY**, indicates that row and column are specified by a x-coordinate and y-coordinate instead.
BLOCK_INCLUDE	is optional; if present must be followed by *Generic_Input_OptKey* that points to a file that contains a set of input that is inserted into the block. *Generic_Input_OptKey* cannot use the keyword **INTERNAL**, since it must reference an external file to include. This feature allows for breaking large data input sections into multiple files that are then inserted into the block. This functions identically to C language `#include "file"` and Fortran `INCLUDE File` statements.

Figure 1.2. —Continued

A

```
BEGIN DUMMY_BLOCK
  #
  Keyword1
  #
  BLOCK_INCLUDE ./Keywords2n3.txt
  #
  # An indented Keyword
             Keyword4   # Comment
END
```

B

```
# Contents of Keywords2n3.txt

Keyword2

Keyword3

# note to self -- remember this note about...
```

C

```
BEGIN DUMMY_BLOCK
Keyword1
Keyword2
Keyword3
Keyword4
END
```

Figure 1.3. Block-Style Input example that defines four input keywords and uses the **`BLOCK_INCLUDE`** keyword to insert two of the four input keywords. *A*, Example block input with the name “DUMMY_BLOCK” that includes two input keywords and inserts the contents from the file “Keywords2n3.txt”. *B*, The contents of the file “Keywords2n3.txt”. *C*, The final version of the block that is read as input by MF-OWHM2. This version removes all blank lines and comments, then shifts to the left any remaining text. [Keyword1, Keyword2, Keyword3, and Keyword4 are example keywords that would be defined for use in the DUMMY_BLOCK block.]

Overview of the List-Array Input—Top-Down View

The *List-Array Input* is a simple input structure that relies on keywords to define the frequency with which input is loaded and the structure of the input. This section gives a broad top-down overview of the *List-Array Input*. First, a flow chart illustrates the keyword decision-guided tree that loads the input. After the flow chart, a summary of all the potential keyword combinations is discussed. This section is meant to be an overview of the input utilities that are defined in detail in the subsequent appendix sections.

Flow Chart

To visualize a top-down view of the *List-Array Input*, the flow chart in appendix figure 1.4 presents the decision tree behind each keyword. It begins with the Temporal keyword that defines the frequency that the input is loaded. The options for the Temporal keyword are **`STATIC`**, **`TRANSIENT`**, or **`CONSTANT`**. The next decision is the input style in which the data are formatted. The two input styles that are supported are a record-based *List Style* input and a model grid shaped *Array Style* input. The *List-Array Input* does support a third input style called *IXJ Style*—which is only mentioned here for completeness and not included in the flow chart nor in subsequent sections. This input style is an advanced input whose structure varies and serves as a surrogate for *List Style* and *Array Style*. The keyword for *List Style* is **`LIST`**, for *Array Style* is **`ARRAY`**, and *IXJ Style* is **`IXJ`**.

Once the input frequency and format are defined, then the actual data location is specified with a *Generic_Input* and loaded with the *Universal Loader* (`ULOAD`). The flow chart uses the word "stress period", corresponding to a typical frequency, but the input load frequency can be different depending how the calling package defines it.

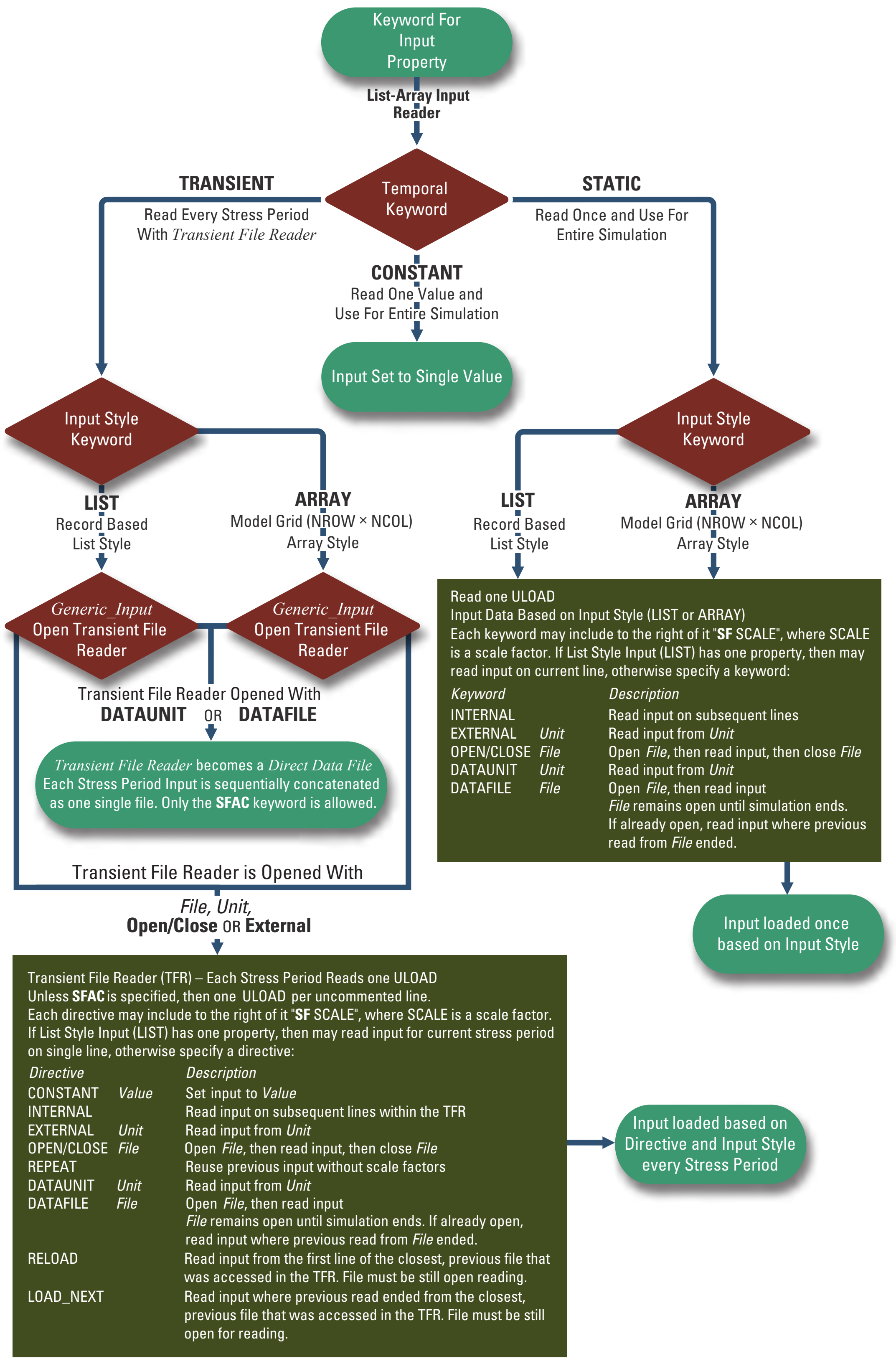
Potential Input Combinations

The *List-Array Input* utility uses keywords as presented in the flow chart (fig. 1.5). The following are the seven keyword combinations and their basic descriptions. The detailed descriptions are provided in the remainder of this appendix.

If the *Transient File Reader* (`TFR`) is used, then it has a set of directive keywords that are specified on each row of its file to indicate each time interval's input (typically one keyword for each stress period, unless noted by the specific MF-OWHM2 input). Note that each uncommented, non-**`SFAC`** row in the *Transient File Reader* is loaded for each time interval. If **`SFAC`** is found (see "**`SFAC`**—Scale Factor Keyword" for more details), then it is applied to the input from the file specified after it. Figure 1.6 presents the supported directive keywords that can be specified within a `TFR`.

Input is loaded from any uncommented line that contains one of the keywords defined in 1–10 in figure 1.6. The input frequency depends on the model feature, but typically is in line with the MODFLOW stress period (for example, the fifth uncommented row containing the keyword in 1–10 serves as input to stress period five). For example, several of the input options in the Farm Process, such as precipitation arrays, can be read by stress period or by time step (the input frequency. The keyword **`SFAC`** may appear multiple times, one per uncommented line, and is only applied to the subsequent input specified by items 1through 10 (that is, it modifies the stress-period input only for that stress period). Figure 1.7 is a simple example `TFR` that loads nine stress periods of input and includes an **`SFAC`** that is applied in the first stress period.

This concludes the top-down overview to the *List-Array Input*; what follows begins the bottom-up development of the utilities that make it up. The abbreviation `LAI` refers to the *List-Array Input* utility input options, and the letters S, T, A, L stand for **`STATIC`**, **`TRANSIENT`**, **`ARRAY`**, and **`LIST`**, respectively. These four options for specification of the data input used for the Farm Process (FMP) block input are explained in detail in the "Farm Process Input Updates" (appendix 6).


Keyword For Input Property
List-Array Input Reader
Temporal Keyword
TRANSIENT
Read Every Stress Period With *Transient File Reader*
STATIC
Read Once and Use For Entire Simulation
CONSTANT
Read One Value and Use For Entire Simulation
Input Set to Single Value
Input Style Keyword
LIST
Record Based List Style
ARRAY
Model Grid (NROW × NCOL) Array Style
Input Style Keyword
LIST
Record Based List Style
ARRAY
Model Grid (NROW × NCOL) Array Style
Generic_Input Open Transient File Reader
Generic_Input Open Transient File Reader
Transient File Reader Opened With
DATAUNIT OR **DATAFILE**
Transient File Reader becomes a *Direct Data File*
Each Stress Period Input is sequentially concatenated as one single file. Only the **SFAC** keyword is allowed.
Transient File Reader is Opened With
File, *Unit*,
Open/Close OR **External**
Input loaded once based on Input Style
Input loaded based on Directive and Input Style every Stress Period

Read one ULOAD
Input Data Based on Input Style (LIST or ARRAY)
Each keyword may include to the right of it "**SF** SCALE", where SCALE is a scale factor. If List Style Input (LIST) has one property, then may read input on current line, otherwise specify a keyword:

Keyword		*Description*
INTERNAL		Read input on subsequent lines
EXTERNAL	*Unit*	Read input from *Unit*
OPEN/CLOSE	*File*	Open *File*, then read input, then close *File*
DATAUNIT	*Unit*	Read input from *Unit*
DATAFILE	*File*	Open *File*, then read input *File* remains open until simulation ends. If already open, read input where previous read from *File* ended.

Transient File Reader (TFR) – Each Stress Period Reads one ULOAD
Unless **SFAC** is specified, then one ULOAD per uncommented line.
Each directive may include to the right of it "**SF** SCALE", where SCALE is a scale factor.
If List Style Input (LIST) has one property, then may read input for current stress period on single line, otherwise specify a directive:

Directive		*Description*
CONSTANT	*Value*	Set input to *Value*
INTERNAL		Read input on subsequent lines within the TFR
EXTERNAL	*Unit*	Read input from *Unit*
OPEN/CLOSE	*File*	Open *File*, then read input, then close *File*
REPEAT		Reuse previous input without scale factors
DATAUNIT	*Unit*	Read input from *Unit*
DATAFILE	*File*	Open *File*, then read input *File* remains open until simulation ends. If already open, read input where previous read from *File* ended.
RELOAD		Read input from the first line of the closest, previous file that was accessed in the TFR. File must be still open reading.
LOAD_NEXT		Read input where previous read ended from the closest, previous file that was accessed in the TFR. File must be still open for reading.

Figure 1.4. Flow chart showing the keyword-based control of new MF-OWHM2 input utilities.

```
1) KEYWORD  CONSTANT   VALUE
2) KEYWORD  STATIC     LIST   ULOAD
3) KEYWORD  STATIC     ARRAY  ULOAD
4) KEYWORD  TRANSIENT  LIST   TFR                 → Transient File Reader
5) KEYWORD  TRANSIENT  ARRAY  TFR                 → Transient File Reader
6) KEYWORD  TRANSIENT  LIST   DATAFILE FILE       → Direct Data File
7) KEYWORD  TRANSIENT  LIST   DATAUNIT UNIT       → Direct Data File
```

where

KEYWORD	is a package input keyword (`PIK`) that identifies the model input feature being set with the *List-Array Input* utility. The keyword depends on the package and the feature being loaded.
CONSTANT `VALUE`	indicates a single value, `VALUE`, is used for the entire simulation.
STATIC	indicates that the input is read once and used for the entire simulation.
TRANSIENT	indicates that the input is read for every time interval (stress period).
LIST	indicates that input uses *List Style*.
ARRAY	indicates that input uses *Array Style*.
`ULOAD`	is the *Universal Loader* that reads input information.
`TFR`	is a *Transient File Reader* that is opened with *Generic_Input_OptKey*. The `TFR` cannot be opened with **INTERNAL**, **DATAFILE**, or **DATAUNIT**.
DATAFILE `FILE`	indicates that the transient input bypasses the `TFR` and instead reads the data directly within the `FILE`. `FILE` is then the location of a *Direct Data File* (`DDF`).
DATAFILE `UNIT`	indicates that the transient input bypasses the `TFR` and instead reads the data directly within the `UNIT` number declared in the NAME file as “ `DATA UNIT FILE` ” or as “ `DATA(BINARY) UNIT FILE` ” where `FILE` is then the location of the `DDF`.
Transient File Reader	is a file that contains a directive keyword for each stress period that indicates where the input is located, then uses `ULOAD` to read the stress period’s input.
Direct Data File	is a single file containing all the input for all stress periods. A `DDF` may only contain the actual input and, optionally, **SFAC** keywords. It is analogous to a `TFR` with only **INTERNAL** directives, but the `DDF` equivalent does not include the keyword **INTERNAL**.

Figure 1.5. Possible keyword combinations for an input that uses the *List-Array Input* utility and a brief explanation about them.

0) **SFAC**	[**DIMKEY**] ULOAD		Optional, advanced scale factor features. Repeat as needed, using one **SFAC** per line of text. The **SFAC**s are multipliers for item 1–10. ULOAD reads a single scale factor, unless a **DIMKEY** keyword is included, which defines the number of scale factors read by ULOAD and how they are applied.
1) **CONSTANT**	VALUE		Set all input to VALUE.
2) **INTERNAL**		[**SF** SCALE]	Input on subsequent lines. **SF** SCALE is optional scale factor, do not include [ ], where SCALE is a number that multiplies with the input. **SF** SCALE may be repeated as needed on the same line
3) **OPEN/CLOSE**	FILE	[**SF** SCALE]	Input within FILE. First open file, then read input from first line, then close file after input is loaded.
4) **EXTERNAL**	UNIT	[**SF** SCALE]	Input is in file UNIT, which defined in the NAME file. First read of UNIT during a simulation loads from the first line. Subsequent references to UNIT read from current line. File UNIT remains open until simulation ends.
5) **DATAUNIT**	UNIT	[**SF** SCALE]	Same as **EXTERNAL**.
6) **DATAFILE**	FILE	[**SF** SCALE]	Check if FILE has been opened previously, either as a UNIT in the NAME file or use of **DATAFILE** FILE in the simulation. If FILE is open, then read input from current line. If not previously opened, then open FILE and read input from first line. FILE remains open until simulation ends and subsequent references to FILE read from current line.
7) **REPEAT**		[**SF** SCALE]	Reuse previously loaded, unscaled input. That is, the input before SCALE or **SFAC** is applied.
8) **RELOAD**		[**SF** SCALE]	Move to the first line, then read input from a file that was, in the TFR, used in the closest, previous **EXTERNAL**, **DATAUNIT**, or **DATAFILE** directive.
9) **LOAD_NEXT**		[**SF** SCALE]	Read input from the current line from a file that was, in the TFR, used in the closest, previous **EXTERNAL**, **DATAUNIT**, or **DATAFILE** directive.
10) **SKIP**			Set input to **0.0** or **0** or an empty string.

Figure 1.6. *Transient File Reader* (TFR) directive keywords that direct how input is loaded for each stress period. Each stress period must contain only one directive defined in items 1 through 10. **SFAC** (item 0) is optional and is only applied to the next directive keyword (items 1 through 10). **SF** SCALE is enclosed in brackets to indicate it is optional and is only applied to the directive with which it appears on the same line. **SFAC** and **SF** SCALE scaling only remain in effect for the directive they are applied to. If a new directive is specified, such as **REPEAT**, the scale factors are not carried forward, such that a **REPEAT** directive will repeat using only the previous, unscaled input and then apply any new scale factors associated with the new directive. [Note that these keywords also work with the *Universal Loader* (ULOAD), but have limited functionality outside of the scope of a TFR.]

```
# TFR Example for reading 9 Stress Periods (SP)
#
# The scale factor keyword, SFAC, can be specified before, after, or
# both before and after the TFR input directive (such as INTERNAL, OPEN/CLOSE, or REPEAT)

SFAC  1.25                 # SP1 input is multiplied by 1.25
SFAC  0.80                 # SP1 input is multiplied by 0.8
INTERNAL                   # SP1 input is read on subsequent lines
   1.0   2.0   3.0
   8.0  16.0  24.0                 # Blank lines are allowed

INTERNAL                   # SP2 input is read on subsequent lines
SFAC  1.1                  # SP2 input is multiplied (scaled) by 1.1
   2.0   4.0   6.0
   8.0  16.0  24.0

REPEAT                     # SP3 input uses SP2 unscaled input (ignores the “SFAC  1.1”)

SFAC  1.25                 # SP4 input is multiplied by 1.25
REPEAT                     # SP4 input uses SP2 input without SP2’s scale factor
                           #
OPEN/CLOSE ./INP.txt       # SP5 input is read from “INP.txt”. File is closed after input read.

DATAFILE   ./DAT.txt       # SP6 input is read from “DAT.txt”.
                           # DAT.txt is opened and input is read from first line
                           # DAT.txt remains open until simulation ends

DATAFILE   ./DAT.txt       # SP7 input is read from “DAT.txt”, which is already open.
                           # DAT.txt reads input from where the SP6 read ended

REPEAT     SF 1.25         # SP8 input uses SP7 input and multiplies it by 1.25

RELOAD                     # SP9 input is read from the start of “DAT.txt”, because
                           # DAT.txt is still open and was the previously used file.
```

Figure 1.7. Example *Transient File Reader* (TFR) file that specifies the location of nine stress periods’ input. The input is assumed to be *Array Style* that consists of 2 rows and 3 columns. [Note that these keywords also work with the *Universal Loader* (ULOAD), but have limited functionality outside of the scope of a TFR.]

Generic Input and Generic Output Files

A generalized file input and output Fortran module was written for MF-OWHM2 to standardize how files are opened and maintained. This is currently only implemented in the new MF-OWHM2 features, but may be incorporated to the older packages in subsequent upgrades and code releases. In this report, any documented input sections using the keywords *Generic_Input* or *Generic_Output* refer to this input. As the names indicate, *Generic_Input* refers to files that are "read only" and are meant to be loaded as input; *Generic_Output* refers to files that are "write only" and are opened for writing MF-OWHM2 output and results.

The sections that follow discuss how to set up a *Generic_Input* or *Generic_Output* file. It begins with some of the beneficial features that include buffering of files with the **`BUFFER`** option and the ability to split output files into multiple parts. An in-depth discussion follows about the difference between a text (ASCII or Unicode UTF-8) file and a MF-OWHM2 binary file. For binary files, methods of loading the data with Python scripts are discussed. Lastly, the formal input to the *Generic_Input* or *Generic_Output* file is presented with examples.

Buffering of Files

One important feature that *Generic_Input* and *Generic_Output* files have is the ability to specify a buffered value in kilobytes (KB)—using the post-keyword "**`BUFFER`** `BUF_SIZE_KB`"—that reserves additional random access memory (RAM) for file operations. Each *Generic_Input* and *Generic_Output* file has its own buffer that results in reduced input and output operations (I/O), leading to faster simulation runtimes at the expense of increasing total RAM usage. For *Generic_Input*, the buffer serves as space where the file preloads for faster input. For the *Generic_Output*, the buffer serves as space where output is written until the buffer is full; then the entire buffer is written to the file, and the buffer begins to refill again. This minimizes file writing to the hard drive for the *Generic_Output*, but the output file is only updated after the buffer is full. For example, if the LIST file has a 1024 kilobyte buffer (post-keyword "**`BUFFER`** `1024`"), then it only updates the actual file in 1024 KB chunks. That is, after MF-OWHM2 writes a total of 1024 KB of text to the LIST file, the user is only then able to see the actual file updates. Empirical tests have shown the fastest performance is for buffer sizes between 32 KB and 1024 KB. By default, all files that are opened in MF-OWHM2 have their buffer set to 32 KB. If it is desired to have immediate writing of output files or no preloading of input files, then the buffer should be set to zero ("**`BUFFER`** `BUF_SIZE_KB`" is set to "**`BUFFER`** `0`"). If a model has a design flaw or input structure that does not trigger a clean error message, that is a runtime Fortran stack error is raised, then it is recommended to zero the buffer for the LIST file to ensure that all its contents are written before the simulation halts.

Splitting Generic Output Files into Parts of the Same Size

For long simulations, output files can grow substantially in size causing issues for post-processors or even viewing the results. To overcome this issue, MF-OWHM2 has the option to split a *Generic_Output* file into a new file once the original reaches a user specified size in megabytes (MB), using the post-keyword "**`SPLIT`** `SPLIT_SIZE_MB`". The file size is checked at the end of each stress period to determine if it exceeds the `SPLIT_SIZE_MB` and needs to be split into a new file. When a file is split, the newly created file has the same file name, but with a number appended to the end of the filename root. The header in the original file is also included in each of the split files created, and when the simulation is restarted, all split files are removed. For example, if the original output file was "MyFile.txt" and the keyword used for it was "**`SPLIT`** `900`", then each time the file size exceeds 900MB, a new file is created with the following naming sequence: MyFile.txt, then MyFile01.txt, and then MyFile02.txt. This is most useful for the LIST file, which can become very large.

Text and Binary Format of Generic Input and Generic Output

The *Generic_Input* and *Generic_Output* both support text (ASCII/Unicode UTF-8) format and binary format. The text format is the default and is a human readable format when opened with any text editor. Most text files are stored in the UTF-8 Unicode format, which is backward compatible with ASCII. All text files read by MF-OWHM2 must use the ASCII or UTF-8 Unicode character encodings. It should be noted that the Microsoft Windows program Notepad.exe saves text files using the UTF-8-BOM format that adds a Byte Order Mark (BOM) binary header to a UTF-8 text file. Fortran does not support reading the UTF-8-BOM encoding format, so this text file format should not be used with any program written in Fortran. MF-OWHM2 does check for a Microsoft Windows style BOM and if found will position the start of the file just past the BOM and process the file using UTF-8 character encoding.

In MF-OWHM2, the binary format uses standard FORTRAN 2003 Unformatted Stream I/O, which may have limited portability among computers. The limited portability is not operating system based (for example, Windows, Linux, or Unix), but is the result of the central processing unit (CPU) "endianness" making the binary CPU dependent. There are two types of binary endianness, referred to as big-endian or little-endian format. Most microprocessors for servers and desktops (x86, x86-64, x64, and AMD64 instruction) use the little-endian, making a binary file portable among them.

Binary files can be read by separate Fortran programs that open the file with the `ACCESS="STREAM"` option. The file can also be loaded using Python structures or Python's Numpy Module with the "numpy.fromfile" method. The file must be loaded with the same variable size for each variable position. For example, if the first value in the binary file is a double precision number and the second variable is an integer, then it must be loaded with a double precision variable followed by an integer variable to load correctly. For Fortran, this method is direct and uses the same naming, but for Python, the user has to specify the "numpy.dtype". When a binary file output is requested in MF-OWHM2, its record structure is written to the LIST file, so the user can reconstruct its output. The records use keywords, described in figure 1.8, to indicate the type of storage.

In addition to the keyword, there could be a dimension to the variable written. This is to accommodate arrays that are written to the binary file. To identify arrays, brackets are used with the dimension enclosed within them (for example, [10] or [5,15]). Within the brackets, the words `NROW`, `NCOL`, and `NLAY` represent the model's number of rows, columns, and layers, respectively. Figure 1.9 presents an example of output that is written to the LIST file to indicate the binary file's record structure.

This example indicates that each record written to the binary file writes the `DATE_START` as 19 characters, `PER` as an integer, `STP` as an integer, `DYEAR` as a double precision, and `DATA` as a double precision array of size `NCOL` by `NROW`. Note that most MODFLOW arrays are stored with the model grid's "column" on the first array dimension rather than the second. That is, computer arrays are specified as [Dimension 1, Dimension 2, Dimension 3, …], so "`DATA[NCOL,NROW]`" indicates that the first array dimension is of size `NCOL` (the number of model columns) and the second array dimension is of size `NROW` (the number of model rows). Figure 1.10 and 1.11 are code examples in Fortran and Python, respectively, necessary to load one record from the figure 1.9 binary-stream output. Fortran can use its native variables, if they are the same type and the file is opened using the options `ACCESS="STREAM"` and `FORM="UNFORMATTED"` options. Conversely, Python 3 requires using the NumPy module; defining the binary structure using "numpy.dtype" and loading the binary data with the "numpy.fromfile" method.

Keyword		Storage	Fortran Type	Python Numpy Type
`(double)`	→	8 bytes,	Double Precision,	`numpy.dtype('float64')`
`(int)`	→	4 bytes,	Integer,	`numpy.dtype('int32')`
`(X char)`	→	X × 1 bytes,	Character(X),	`numpy.dtype('SX')`
`(sngl)`	→	4 bytes,	Real,	`numpy.dtype('float32')`

Figure 1.8. Definition of keywords used by MF-OWHM2 to define a stream-binary file's memory storage for different variable types and the type of Fortran and Python variables that can read them. [X is used as a place holder for an integer number, such as Character(19) or numpy.dtype('S19'). NumPy is a library for the Python programming language. Fortran assumes that REAL and DOUBLE PRECISION are not modified by compiler options. To make Fortran compiler option independent, it is recommended to use the parameters INT32, REAL32, and REAL64 from the intrinsic module ISO_FORTRAN_ENV and declare INTEGER variables as INTEGER(INT32), REAL variables as REAL(REAL32), and DOUBLE PRECISION variables as REAL(REAL64).]

```
DATE_START (19char), PER (int), STP (int), DYEAR (double), DATA[NCOL,NROW] (double)
```

Figure 1.9. Example output, written to the LIST file, that indicates the structure of a MF-OWHM2 binary-stream output file. The text not enclosed in parenthesis is the variable name and the part in the parenthesis indicates the binary-stream format used. [Note that MODFLOW stores arrays as NCOL by NROW. 19char indicates the record is a Fortran character of length 19, int indicates the record is a 4-byte Fortran integer (INT32), double indicates the record is an 8-byte Fortran floating point number (REAL64), which is commonly called double precision.]

```
PROGRAM READ_BINARY
USE, INTRINSIC:: ISO_FORTRAN_ENV, ONLY: REAL32, REAL64

! REAL32 is single precision -> REAL(REAL32) ⇔ REAL             (4 bytes)
! REAL64 is double precision -> REAL(REAL64) ⇔ DOUBLE PRECISION (8 bytes)

INTEGER, PARAMETER:: NROW = 5, NCOL = 10  ! Dimension of Model Grid

CHARACTER(len=19):: DATE_START
INTEGER:: PER, STP
REAL(REAL64):: DYEAR
REAL(REAL64), DIMENSION(NCOL, NROW):: DATA  ! MODFLOW Stores Arrays as NCOL x NROW

INTEGER:: IU  ! Variable holds Fortran unit number associated with binary file

IU = 0        ! Initialize the variable, will be set by OPEN with Fortran unit number

OPEN(NEWUNIT=IU, FILE="myfile.bin", STATUS="OLD", ACCESS="STREAM", FORM="UNFORMATTED")

! Read one binary record from file unit IU

READ(IU) DATE_START, PER, STP, DYEAR, DATA

END PROGRAM
```

Figure 1.10. Fortran code example that can read a stream-binary file “myfile.bin” that contains binary record composed of 19 characters, 2 integers, a double-precision real number, and then a double precision array of size that is 5 model rows and 10 model columns. [Note that MODFLOW stores arrays as NCOL by NROW, where NROW is 5 and NCOL is 10. DATE_START is the calendar date at the start of the time step, PER is the stress period number, STP is the time step number, DYEAR is a decimal year representation of DATE_START, DATA is the NCOL by NROW array that is read.]

```python
import numpy

NROW = 5
NCOL = 10

# Note that MODFLOW stores binary arrays as NCOL x NROW

# But Python stores arrays by row, while Fortran stores array by column,
#   so unlike a Fortran read, Python reads NROW x NCOL
#   and then must transpose the result to get the original shape (NCOL, NROW)

dt = numpy.dtype([
                 ('DATE_START', numpy.dtype('S19')                    ),
                 ('PER',        numpy.dtype('int32')                  ),
                 ('STP',        numpy.dtype('int32')                  ),
                 ('DYEAR',      numpy.dtype('float64')                ),
                 ('DATA',       numpy.dtype('float64'), (NROW, NCOL) ),
                 ])

# Set "F" to binary file that will be read from

F = open('myfile.bin','rb')

# Read single record defined with dt from file "F"

REC = numpy.fromfile(F, dtype=dt, count=1)

# REC holds each record as an array of length 1,
#   the following extracts the binary contents to individual variables

DATE_START = REC['DATE_START'][0]        # Get stored starting date
PER        = REC['PER'       ][0]        # Get stress period number
STP        = REC['STP'       ][0]        # Get time step number
DYEAR      = REC['DYEAR'     ][0]        # Get starting date decimal year
DATA       = REC['DATA'][0].transpose()  # Get DATA as NCOL x NROW array
```

Figure 1.11. Python 3 code example that can read a stream-binary file “myfile.bin” that contains binary record composed of 19 characters, 2 integers, a double precision real number, and then a double precision array of size that is 5 model rows and 10 model columns. [Note that MODFLOW stores arrays as NCOL by NROW, where NROW is 5 and NCOL is 10. Python reads arrays using the right most dimension first, whereas Fortran reads the left most dimension first. Because of this, the DATA array, which was written with Fortran as NCOL by NROW, is read by Python with an array dimensioned as NROW by NCOL, then must be transposed to get the original MODFLOW array. DATE_START is the calendar date at the start of the time step, PER is the stress period number, STP is the time step number, DYEAR is a decimal year representation of DATE_START, DATA is the NROW by NCOL array that is read, REC holds the first record in myfile.bin read by numpy.fromfile.]

Input Structure

The *Generic_Input* and *Generic_Output* module systematically looks for keywords to indicate how to access or open a file and what options to include with it. When keywords are optional (fig. 1.12*A*, items 1 and 7), then the "_OptKey" is added to *Generic_Input* and *Generic_Output*, changing it to *Generic_Input_OptKey* or *Generic_Output_OptKey.* Because of the potential for ambiguity of input options, it is recommended to use the keywords even when they are optional.

The order in which the *Generic_Input_OptKey* and *Generic_Output_OptKey* module detects and opens a specified file is to first check to see if it can load a single integer; if so, then the integer indicates that it is a unit number that is defined in NAME file, which is associated with a DATA or DATA(BINARY) file (fig. 1.12*A*, item 1). If it fails to read an integer, then it checks to see if there are keywords for the unit number; keywords to open a file; or lastly, if the line just contains a file name to open (fig. 1.12*A*, items 2 through 7). Once the file has been identified through its unit number or file name, there are a set of optional post-keywords and scale factors available (fig. 1.12*A*, items A through H), that override default options or allow advanced file operations. Any comments to the right and on the same line must be preceded by a "**#**" symbol (commonly called a number sign, pound sign, or hash symbol). Preceding a comment by "**#**" symbol is necessary to indicate to MF-OWHM that the text to the right is a comment and not a post-keyword. Figure 1.12*A* contains the decision order for how the file is detected, opened, and what post-keywords can be applied, and a description of the keywords is in figure 1.12*B* and figure 1.12*A*, *C*. Because of the potential for ambiguity of input, it is recommended to use the keywords **`EXTERNAL`** and **`OPEN/CLOSE`** (fig. 1.12) rather than directly loading the `UNIT` or `FILE` name for the *Generic_Input_OptKey* and *Generic_Output_OptKey* versions.

The two most powerful optional keywords for the *Generic_Input* are the **`SF`** `SCALE` and **`REWIND`**. When **`SF`** `SCALE` is loaded, it is multiplied by any data loaded from the *Generic_Input*. Multiple scale factors are allowed to provide clarity for the input. For example, two scale factors could be used to separate a unit conversion factor from a calibration factor. Continuing this example, an input line that reads from the file MyText.txt could be "**`OPEN/CLOSE`** `MyFile.txt` **`SF`** `0.3048` **`SF`** `1.05`", where `0.3048` is a unit conversion factor and `1.05` is a calibration adjustment factor that increases the input by 5 percent. The **`REWIND`** option resets a file that is already opened (excluding **`INTERNAL`**) to the first line. This is advantageous if a file only has a certain number of input records that are repeated after a set number of stress periods. For example, if an input was repeated every 12 stress periods (that is, one input for each month), then it only requires 12 lines to represent the 12 months in one file that is opened with **`DATAFILE`**, **`DATAUNIT`**, or **`EXTERNAL`**. Once the file is accessed and read 12 times for the 12 stress periods, the file utility then uses the keyword **`REWIND`** to move to the start of the file to cycle through another 12 stress periods.

The following is a set of examples for different ways to access or open a file using *Generic_Input* or *Generic_Input_OptKey*:

```
INTERNAL    # READ input on subsequent lines
```

```
# READ input on subsequent lines, multiply input loaded by 2.5 and 1.25
INTERNAL  SF 2.5   SF 1.25
```

```
# READ input on subsequent lines, multiply input loaded by 2.5 and 1.25
INTERNAL  2.5  1.25
```

```
# READ input on subsequent lines, multiply input loaded by 2.5, "#" excludes 1.25
INTERNAL  2.5  # 1.25
```

```
OPEN/CLOSE  MyFile.txt  # Open MyFile.txt, load its contents, then close file
```

```
# Open MyFile.txt, load its contents and multiply it by 2.5 and 1.25, then close file
OPEN/CLOSE  MyFile.txt    SF 2.5  SF 1.25
```

```
# Open MyFile.txt, load its contents and multiply it by 2.5 and 1.25, then close file
OPEN/CLOSE  MyFile.txt        2.5      1.25
```

```
EXTERNAL 55    # Load contents from Unit 55 as specified in the NAME file
```

```
# Load contents from Unit 55 as specified in the NAME file, multiply it by 2.5 and 1.25
EXTERNAL 55   SF 2.5  SF 1.25
```

```
# Load contents from Unit 55 as specified in the NAME file, multiply it by 2.5 and 1.25
EXTERNAL 55      2.5     1.25
```

The following are additional examples of different ways to access or open input with *Generic_Input_OptKey*:

```
MyFile.txt     # Open MyFile.txt, load its contents, then close file
```

```
# Open MyFile.txt, load its contents and multiply it by 2.5 and 1.25, then close file
MyFile.txt   SF 2.5  SF 1.25
```

```
# Open MyFile.txt, load its contents and multiply it by 2.5 and 1.25, then close file
MyFile.txt      2.5     1.25
```

```
55             # Load contents from Unit 55 as specified in the NAME file
```

```
# Load contents from Unit 55 as specified in the NAME file, multiply it by 2.5 and 1.25
55   SF 2.5  SF 1.25
```

```
# Load contents from Unit 55 as specified in the NAME file, multiply it by 2.5 and 1.25
55      2.5     1.25
```

The following is a set of examples for different ways to specify keywords for *Generic_Output* or *Generic_Output_OptKey*:

```
INTERNAL                 # Write output to LIST file
```

```
LIST                     # Write output to LIST file, same as INTERNAL
```

```
OPEN/CLOSE MyFile.txt # Open MyFile.txt and write output to it
```

```
EXTERNAL 55    # Write output to Unit 55 as specified in the NAME file
```

The following are additional examples for different ways of setting up the output for *Generic_Output_OptKey*:

```
MyFile.txt     # Open MyFile.txt and write output to it
```

```
55             # Write output to Unit 55 as specified in the NAME file
```

The following are examples for present the usage of post-keywords for *Generic_Output and Generic_Output_OptKey*:

```
OPEN/CLOSE MyFile.txt       SPLIT 500  #MB
```

```
# Write output to Unit 55 and split file every 500MB. Note if file is binary,
# then it should be declared as DATA(BINARY) in NAME file
55 SPLIT 500
```

The second to last example would split the *Generic_Output* file “MyFile.txt” if its size exceeded 500 megabytes and begin writing output to a new file called “MyFile01.txt”. If this new file size exceeded 500 megabytes, then it would be split, and output would be written to the file “MyFile02.txt”. This splitting of files continues until the MF-OWHM2 simulation is completed. The following is an example of buffering a file with 512 kilobytes of memory:

```
# This works for Generic_Input too
OPEN/CLOSE MyFile.txt       BUFFER  512  #KB buffer for file
```

Both **BUFFER** and **SPLIT** can be used simultaneously, and the order does not matter. This allows for a file to be buffered and split into multiple files. The following is an example of using both keywords, which would turn off buffering of a binary file, but still split the file whenever it was greater than 500 megabytes in size:

```
# Open MyFile.txt as binary, with no buffer and split to new file every 500MB
MyFile.txt  BINARY  BUFFER 0  SPLIT 500
```

or

```
OPEN/CLOSE MyFile.txt  SPLIT 500  BINARY   BUFFER 0
```

Finally, the keyword **BINARY** may be placed at the start of a *Generic_Input* or *Generic_Output* input section:

```
BINARY OPEN/CLOSE MyFile.txt  SPLIT 500  BUFFER 0
```

A

- One of the following items must be present accessing or opening a file with *Generic_Input*, *Generic_Output*, *Generic_Input_OptKey*, or *Generic_Output_OptKey*:

1)	`UNIT`	→	*Generic_Input_OptKey*	and	*Generic_Output_OptKey*
2)	**`INTERNAL`**				
3)	**`EXTERNAL`** `UNIT`				
4)	**`DATAUNIT`** `UNIT`				
5)	**`OPEN/CLOSE`** `FILE`				
6)	**`DATAFILE`** `FILE`				
7)	`FILE`	→	*Generic_Input_OptKey*	and	*Generic_Output_OptKey*
8)	**`NOPRINT`**	→	*Generic_Output*	and	*Generic_Output_OptKey*

- The following are optional, post-keywords that are checked for after items 1–7.
  The order of the post-keywords does not matter nor do any have to be specified.
  Any option supported by *Generic_Input* is supported by *Generic_Input_OptKey*.
  Any option supported by *Generic_Output* is supported by *Generic_Output_OptKey*.

A)	**`BINARY`**	→	*Generic_Input*	and	*Generic_Output*
B)	**`BUFFER`** `BUF_SIZE_KB`	→	*Generic_Input*	and	*Generic_Output*
C)	**`SPLIT`** `SPLIT_SIZE_MB`	→			*Generic_Output*
D)	**`NOPRINT`**	→			*Generic_Output*
E)	**`REWIND`**	→	*Generic_Input*		
F)	**`DIM`** `DIM_SIZE`	→	*Generic_Input*		
G)	**`SF`** `SCALE`	→	*Generic_Input*		
H)	`SCALE`	→	*Generic_Input*		

Figure 1.12. Syntax for accessing or opening files with the utilities: *Generic_Input*, *Generic_Output*, *Generic_Input_OptKey*, and *Generic_Output_OptKey*. *A*, The keyword decision order for how the file is detected, then either accessed or opened (items 1 through 8), and what post-keywords can be applied (items A through H). *B*, Explanation of items 1 through 8 in part *A*. *C*, Explanation of the post-keywords in part *A*. [The symbol, → indicates that the option is only available to a specific file utility.]

B

INTERNAL	indicates that *Generic_Input* files are read on subsequent lines, and *Generic_Output* files are written to the LIST file.
NOPRINT	indicates that *Generic_Output* output will not be written.
`UNIT`	is the unit number of a file opened in the NAME file with `DATA` or `DATA(BINARY)`.
EXTERNAL	indicates that *Generic_Input* files are read from a file associated with `UNIT`, and *Generic_Output* files are written to a file associated with `UNIT`.
DATAUNIT	This option is identical to **EXTERNAL**, except if the expected file to be opened is a *Transient File Reader* (`TFR`), then **DATAUNIT** indicates that the `TFR` is bypassed the `UNIT` is a *Direct Data File* (`DDF`). A `TFR` and `DDF` are part of the *List-Array Input* (`LAI`).
BINARY	is a keyword that indicates `FILE` should open as a **BINARY** file. The keyword may be placed at either the beginning or ending of the *Generic_Input* and *Generic_Output* input.
`FILE`	is the file name and location (file path) that will be opened for reading or writing to. Note that item 7 is equivalent to specifying **OPEN/CLOSE** `FILE`.
OPEN/CLOSE	indicates that `FILE` is to be opened and read or written to at the start of the file. `FILE` is closed when it is no longer required. *Generic_Input* file is closed after reading input. Note, a `TFR` reads input until the simulation ends. *Generic_Output* is closed when either the simulation ends or it is no longer used for writing output.
DATAFILE	indicates that if `FILE` is not open, then open it; The file remains open until simulation ends. Any subsequent **DATAFILE** references to the same `FILE` continue reading or writing at the file's previous position. This is analogous to **EXTERNAL** `UNIT`, but uses the file name, `FILE`, instead of `UNIT`, and it is not required to define `FILE` in the NAME file. If the expected file to be opened is a `TFR`, then **DATAFILE** indicates that the `TFR` is bypassed and `FILE` is a `DDF`.

Figure 1.12. —Continued

C

`BUFFER`	indicates that the input or output file should be buffered in RAM to improve speed. The size of the buffer is in kilobytes (KB). When not specified the default buffer is 32KB. *Generic_Input* preloads the file into RAM for reading. *Generic_Output* writes output first to RAM, then writes to the actual file in `BUF_SIZE_KB` chunks.
`BUF_SIZE_KB`	Size of the buffer in KB. If the buffer is set to `0`, then *Generic_Input* disable preloading of the file before reading. *Generic_Output* results in immediate writing of output.
`SPLIT`	indicates that a *Generic_Output* file should split into a new file after a specified size in megabytes (MB). The new file uses the original files name with a sequential number appended to it to not overwrite the original.
`SPLIT_SIZE_MB`	is the minimum size, in MB, of the output before it is split.
`REWIND`	indicates that the file's position is reset to the start of the file before the file is read from or written to This is done by default for newly opened files. This option is ignored when used with the keyword **`INTERNAL`**.
`DIM`	sets user-specified dimension, `DIM_SIZE`, for the input file. The use of `DIM` depends on what the input file is used for. If the input file does not support `DIM`, then a warning is raised. One notable use is that if `DIM_SIZE` is specified when opening a *Transient File Reader* (`TFR`) and *Direct Data File* (`DDF`), then it specifies the largest line size that is read as input and overrides the default size of 700 characters.
`DIM_SIZE`	is an integer number that is specified after **`DIM`**.
`SF`	a scale factor, `SCALE`, is read and multiplied with the input data. "**`SF`** `SCALE`" may be repeated multiple times. Scale factors are ignored for input that is expected as an integer (`int`).
`SCALE`	is a scale factor that is read and multiplied with the input data. The use of the keyword **`SF`** is optional but recommended for clarity. "`SCALE`" may be repeated multiple times.
`NOPRINT`	suppresses the creation and writing of an output file.

Figure 1.12. —Continued

ULOAD Input Utility and SFAC Keyword—Universal Loader and Scale Factors

The following section introduces the *Universal Loader* (ULOAD) input utility and optional keyword **SFAC**, which provides supplemental scale factors for modifying input. ULOAD is an array reading utility that is capable of loading text, integers, and floating-point numbers that are structured as *List Style* input or *Array Style* input. *List Style* is a record-based input that reads one record per row; a single record starts with a record identifier (ID, an integer) and is followed by one or more columns of input data. *Array Style* loads a two-dimensional array of input data without a record ID (identifier). *List Style* may have its record ID associated with a specific input property or may be linked to a spatial identification array that is loaded with the *Array Style*. Typically, *Array Style* is used for loading spatial data that conform to the MF-OWHM2 model grid (NROW by NCOL). Ultimately, the key difference between the *List Style* and *Array Style* is whether record IDs are read or not.

Note that MODFLOW, and consequently MF-OWHM2, reads (or writes) arrays from (or to) text files with the dimension NROW by NCOL (traditional matrix structure), but reads (or writes) arrays from (or to) **BINARY** files with the dimension NCOL by NROW. This is the reason why figures 9 and 10 dimension the array as (NCOL, NROW), which is an array that is read from a binary file.

ULOAD—A Universal Array and List Load Utility

The MF-OWHM2 ULOAD input utility is capable of loading either *List Style* or *Array Style* input data. The utility automatically skips blank lines and ignores any commented text, which must be preceded with "#" symbol. The data loaded depend on the input keywords specifying what the dataset comprises, which can be either integer, floating point (single/double precision), or character data (for example, ASCII text). The specific data type is defined by the input package that is relying on ULOAD to load the data.

List Style reads a row at a time that starts with a record ID (as an integer) then multiple records—properties—to the right of it. The *Array Style* loads a two-dimensional array (typically, NROW by NCOL) and must be formatted in the structure of the array (for example, it must have NCOL inputs for each row and must have NROW rows). The input data may be space separated, comma separated, or tab separated, and comments are allowed between rows and after the last column of input. Figure 1.13 presents two *List Style* input examples and one and *Array Style* example.

Unless otherwise stated in the input section of the utility that is calling ULOAD, the row record ID is the only placeholder not used by MF-OWHM2. It is instead required to provide an ID to help the user identify the input ID (or row of input). In addition, unless otherwise stated by the input section, the rows must be well ordered, irrelevant of the ID, such that the first row applies to record 1, and the second row applies to record 2, and so forth. The number of rows that are read is specified by each input section.

ULOAD is capable of loading binary files. This is an advanced input and is discussed here only to provide a complete description of ULOAD. ULOAD imposes several limitations to the structure of the binary file. First, the *Generic_Input* must include the post-keyword **BINARY** or be associated with a unit that was opened with DATA(BINARY) in the NAME file. Second, the *List Style* input cannot have the record ID in the binary file. The binary structure using *List Style* input is such that the first property (from the first record to the last record) is written first, then the second property, and so forth. Third, the *Array Style* binary file array structure will be defined by the input utilizing it. If there is no description of the input structure defined, the *Array Style* input assumes that the array is written in binary format such that the first column is first, then the second column, and so forth. This assumption coincides with the way MODFLOW stores arrays as (NCOL, NROW). The advantage of binary files compared to text files is that they load faster and are typically smaller in size. Because text files are easier to maintain and more portable, however, they are the recommended input format.

A special case is ULOAD_NoID, which follows the same rules, but does not read the row record ID (only the data are read in). In fact, *Array Style* uses ULOAD_NoID to read a two-dimensional array. In addition to this, ULOAD_NoID is how a single value (for example, a single integer, scalar or floating-point number, or single word) is loaded. ULOAD_NoID can be thought of as a generalization of the standard MODFLOW input utilities *U1DREL*, *U2DREL* and *U2DINT*, because it performs the same operation without requiring format codes and automatically adjusts for single precision and integer input (and for some special inputs can read text input, such as a name).

The ULOAD input data type (integer, floating point, text) and input style is specified by the calling package, and within the model input dataset being loaded. ULOAD first attempts to identify a keyword (fig. 1.14*A*, items 1 through 4). If ULOAD does not identify a keyword, then the input is assumed to be specified as an *Implied_Internal* and located along the current line. If an *Implied_Internal* is not allowed, then an error is raised, and the simulation stopped.

The following is a set of examples that use `ULOAD` to read in three values (`Val1`, `Val2`, `Val3`) that define one input property (for example, root depth for `NCROP` = 3 Crops). The first example is for using a constant value:

```
CONSTANT VALUE          # Sets Val1 Val2 Val3 Equal to VALUE
```

The **`INTERNAL`** keyword requires a record ID:

```
INTERNAL
ID1  Val1
ID2  Val2
ID3  Val3
```

If no keyword is present and an *Implied_Internal* is allowed, then the input is loaded along the same line, without the List Style record ID. The following is an example *Implied_Internal*:

```
Val1 Val2 Val3   # Load Val1 Val2 Val3 to Record ID 1, 2, 3, respectively
```

The input can be specified in a separate file. One method of loading input is to use the keyword **`OPEN/CLOSE`**:

```
OPEN/CLOSE MyFile.txt
```

The file “MyFile.txt” contains the following:

```
ID1  Val1
ID2  Val2
ID3  Val3
```

The final example uses **`EXTERNAL`**:

```
EXTERNAL 55
```

The entry in the NAME file (NAM) is as follows:

```
DATA 55 ./MyFile.txt   READ BUFFER 16
```

The file “MyFile.txt” contains the following:

```
ID1  Val1
ID2  Val2
ID3  Val3
```

For these examples, values of `ID1`, `ID2`, and `ID3` would be 1, 2, and 3 to serve as the place holder, and `Val1`, `Val2`, and `Val3` would be the input property (for example, 3.14, 2.718, and 1.618).

As described before, ULOAD is capable of loading multiple properties per record. In this case, ULOAD does not allow for an *Implied_Internal* because of the additional column dimensions. The following is an example of a ULOAD with three records that load property A and B.

```
INTERNAL
ID1  Val1_A  Val1_B
ID2  Val2_A  Val2_B
ID3  Val3_A  Val3_B
```

Because ULOAD uses *Generic_Input* to check for the keywords **INTERNAL**, **EXTERNAL**, and **OPEN/CLOSE**, it also supports all the post-keywords (for example, **BUFFER**, **BINARY**, **SF** SCALE). To illustrate the application of **SF** SCALE*,* the following example loads the values in bold and then multiplies them by SCALE1 and SCALE2, which represent multiple instances of **SF** SCALE.

```
INTERNAL
ID1  Val1_A  Val1_B
ID2  Val2_A  Val2_B
ID3  Val3_A  Val3_B
```

or

```
INTERNAL
ID1  Val1_A  Val1_B
ID2  Val2_A  Val2_B
ID3  Val3_A  Val3_B
```

```
INTERNAL  SF SCALE1  SF SCALE2
ID1  Val1_A  Val1_B
ID2  Val2_A  Val2_B
ID3  Val3_A  Val3_B
```

A

```
# List Style Input; 3 IDs and 1 Properties
# ID  Prop1
  1  Val        # Comment
  2  Val
  3  Val
```

B

```
# List Style Input; 3 IDs and 2 Properties
# ID  Prop1  Prop2
  1  Val  Val # Comment
  2  Val  Val
  3  Val  Val
```

C

```
# Array Style Input; 4 rows and 5 columns
 Val  Val  Val  Val  Val # Comment
 Val  Val  Val  Val  Val
 Val  Val  Val  Val  Val
 Val  Val  Val  Val  Val
```

Figure 1.13. Example input structures expected by *Universal Loader* (ULOAD). *A*, *List Style* input that reads three records that define one property. *B*, *List Style* input that reads three records that each define two properties. *C*, *Array Style* input that reads a 4-by-5 array. [Val is a placeholder that represents a single value of the format expected by the input using ULOAD.]

A

0) **SFAC** [DIMKEY] ULOAD	Optional and may be repeated with one **SFAC** per line. Defines advanced scale factors that are applied to input indicated by items 2–5. **SFAC** scale factors are read with a separate instance of ULOAD. If present, then item 2, 3, 4, or 5 is expected on the next uncommented line.
1) **SKIP**	
2) **REPEAT** [**SF** SCALE]	Only allowed when the same input was previously loaded with ULOAD. Typically, this only occurs within a *Transient File Reader*.
3) **CONSTANT** VALUE	
4) *Generic_Input* [**SF** SCALE]	Specify the location of the input with **INTERNAL**, **EXTERNAL**, **OPEN/CLOSE**, **DATAUNIT**, or **DATAFILE**. At the start of the input within the *Generic_Input* "**SFAC** [DIMKEY] ULOAD" can optionally be included.
5) *Implied_Internal*	Input is assumed to be on current line if and only if the expected input is a single VALUE, or *List Style* input with only one property, or *Array Style* input that is a vector (one-dimensional array).

Figure 1.14. *Universal Loader* (ULOAD) supported keywords. *A*, The syntax of ULOAD keywords, ordered by how ULOAD checks for keywords; for example, **REPEAT** is checked for before **CONSTANT**. *B*, Explanation of items 0 through 5 in part *A*. [**SF** SCALE is enclosed in brackets to indicate it is optional and is only applied to the directive with which it appears on the same line. **SFAC** and **SF** scaling only remain in effect for the directive they are applied to. If a new directive is specified, such as **REPEAT**, the scale factors are not carried forward, such that a **REPEAT** directive will by default repeat only the unscaled input.]

B

SFAC [**DIMKEY**] ULOAD

SFAC indicates that advanced scale factors are read with a separate instance of ULOAD and applied to the input read by items 2–5.

SFAC may be repeated as necessary, but only one is allowed per line.

DIMKEY is an optional keyword that indicates the number of scale factors read and how they are applied to the input.

If **DIMKEY** is not present or not supported,
then **SFAC** reads a single scale factor that is applied to all the input.

If **DIMKEY** is supported, then the accepted keywords and how they are applied to the input are defined by the input that supports it.

SKIP specifies that values input properites are set to zero.

REPEAT specifies input that was previously loaded should be reused.

Repeated input does not carry forward any **SFAC**, **SF**, or SCALE scale factors from the previous load;
thus it requires the scale factors to be repeated to reproduce the same input.

REPEAT is only allowed if the dataset has been previously loaded by ULOAD (that is, the input from the previous stress period).

SF SCALE specifies an optional scale factor, SCALE, that is applied to the **REPEAT** input or the input read from the *Generic_Input*.
Multiple **SF** SCALE are allowed but must be on the same line.
SCALE, without the keyword **SF**, is accepted in the place of "**SF** SCALE".

CONSTANT specifies that VALUE should be used as to the input.
VALUE must match the data type that the input utility expects,
which can be integer, floating point, or text.

Generic_Input uses the keywords **INTERNAL**, **EXTERNAL**, **OPEN/CLOSE**, **DATAUNIT**, and **DATAFILE** to specify the location of the input.

All the *Generic_Input* post-keywords are supported, including **SF** SCALE.

SFAC [**DIMKEY**] ULOAD can be used within the *Generic_Input* on the line (or lines) before the expected input data begins.

Implied_Internal assumes that the input data is located along the current line because items 1–4 are not identified.

An error is raised if input is not one of the following cases:

- A scalar—that is a single input value.
- *List Style* input that only has one property (record ID and one column)
- *Array Style* input that expects a vector (one-dimensional array)

A single property *List Style* input in an *Implied_Internal* does not include the record ID and assumes that the first input has a record ID of 1, and the second has a record ID of 2, and so forth.

Figure 1.14. —Continued

SFAC—Scale Factor Keyword

MF-OWHM2 input sections may specify that they support **SFAC** as a keyword. **SFAC** allows loading and applying scale factors to a set of input data. **SFAC** is automatically multiplied with any loaded *Generic_Input* "SCALE" values to derive a composite scale factor. **SFAC** scale factors rely on a *List Style* ULOAD (see previous section) to read either a single scale factor or a set of scale factors. If **SFAC** reads a set of scale factors, then it is required to include the keyword **DIMKEY** to specify the number of scale factors and how they should be applied. Figure 1.15 presents the general input structure when using **SFAC**.

Because **SFAC** relies on ULOAD with *List Style* input to read the scale factors, it must include a record ID. If the scale factors are read with an *Implied_Internal*, then the record ID is not read. For simplicity of input, it is recommended to use the *Implied_Internal* for scale factors.

The following are examples of ways to load scale factors for an input utility that accepts a **DIMKEY** of "**ByWBS**" and has three water balance subregions (WBS or farms with NWBS = 3). Note that **SFAC** always accepts the **DIMKEY** keyword **ALL**, but it is optional because **SFAC** automatically reads a single scale factor if **DIMKEY** is not present.

The following examples load a single scale factor, SVAL, that is multiplied by whatever input it is associated with. The first two text boxes use a ULOAD *Implied_Internal* to load SVAL, and the second two boxes use ULOAD with a *Generic_Input* keyword to specify where to find SVAL—note ULOAD requires the record ID:

```
SFAC SVAL
```

or

```
SFAC ALL SVAL
```

or

```
SFAC INTERNAL
ID1    SVAL
```

or

```
SFAC OPEN/CLOSE  MyText.txt
```

MyText.txt contains the following:

```
ID1    SVAL
```

The following illustrate using ULOAD with an *Implied_Internal* and two *Generic_Input* keyword examples that load the scale factors SVAL1, SVAL2, SVAL3 to be applied where WBS 1, 2, and 3 are located, respectively:

```
SFAC ByWBS  SVAL1 SVAL2 SVAL3
```

```
SFAC ByWBS INTERNAL
ID1  SVAL1
ID2  SVAL2
ID3  SVAL3
```

```
SFAC ByWBS OPEN/CLOSE  MyText.txt
```

MyText.txt contains the following:

```
ID1  SVAL1
ID2  SVAL2
ID3  SVAL3
```

Because a *List Style* `ULOAD` is used to load the **`SFAC`** scale factors, this method also supports loading additional `SCALE` factors by using the *Generic_Input* post-keyword. This is useful for keeping conversion factors or correction factors separated from calibration scale factors.

The following is an example that includes a conversion factor of inches to meters and a correction factor of 0.99 that are multiplied by all the scale factors in MyText.txt. The product is then multiplied by the input data (the following example uses real numbers to clarify the multiplication):

```
SFAC ByWBS OPEN/CLOSE MyText.txt  0.0254  0.99
```

MyText.txt contains the following:

```
1  1.5
2  2.5
3  3.5
```

In this example in the input, 0.0377 (that is, 0.0254 × 0.99 × 1.5), is multiplied by the associated input property everywhere WBS 1 resides, 0.0629 is the multiplier everywhere WBS 2 resides, and 0.088 is the multiplier everywhere WBS 3 resides.

`SFAC` allows for a layered approach to scale factors that adds flexibility for calibration and distinguishes conversion factors from parameter-estimation factors. For example, the file MyText.txt could be a calibration template file that optimization software uses to modify the MF-OWHM2 input, but the other scale factors do not change because they just transform the input to the proper units.

```
SFAC  [DIMKEY]  ULOAD
```

where

`SFAC`	is the keyword that initiates the advanced scale factor routine. Only one **`SFAC`** is allowed per line
`DIMKEY`	is an optional keyword that indicates the number of scale factors to be read in and how they are applied. If **`DIMKEY`** is not present or is set to "**`ALL`**", then a single scale factor is read and applied to the input data. An example **`DIMKEY`**, from FMP, is "**`ByWBS`**", which indicates **`NWBS`** scale factors are read, and the first scale factor is applied to areas where WBS 1 is located, and the second scale factor is applied where WBS 2 is located, and so on.
`ULOAD`	is *Universal Loader* utility that reads the scale factors with *List Style* input. `ULOAD` can use an *Implied_Internal* to read the scale factors, which is recommended method for reading **`SFAC`** scale factors. It is not recommended to use keyword **`INTERNAL`** to load scale factors.

Figure 1.15. Advanced scale factor input structure and explanation of keywords.

ULOAD That Contains SFAC

Any input utility file that is loaded with a call to ULOAD also supports the **SFAC** keyword. This can be confusing because **SFAC** relies on a separate instance of ULOAD to load the scale factors. If there is a single **SFAC** in a ULOAD, the routine ULOAD is called twice; the first time loads the scale factor data, and the second time loads the input data and applies the scale factors to it.

ULOAD supports multiple calls of **SFAC**, but the keyword **SFAC** must be located before the input data and only have one **SFAC** keyword for each line. The exception to this is when reading *List Style* with *Implied_Internal*, ULOAD will check for a single **SFAC** keyword after the list records read on the same line. If an input record read by ULOAD uses a *Generic_Input* keyword (for example, **INTERNAL**, **EXTERNAL**, **OPEN/CLOSE**, **DATAUNIT**, and **DATAFILE**), then it also checks in the *Generic_Input* file for the **SFAC** keyword before loading the input data. This allows for scale factors to be applied before checking for the *Generic_Input* flags and in the *Generic_Input* file itself. In addition to this, if the *Generic_Input* is followed by a SCALE value that is written to the right of it, it is also included as a scale factor for the data.

The following examples use **SFAC** with ULOAD to load three values called Val1, Val2, and Val3 and two scale factors called SF1 and SF2. The general form, figure 1.16, of this load has one or two "**SFAC** ULOAD" to read a single scale factor and then reads one ULOAD that loads Val1, Val2, and Val3.

This example loads the scale factor SF1, then uses an *Implied_Internal* on the next line to load the three values:

```
SFAC SF1          ← Implied_Internal read of SF1
Val1 Val2 Val3    ← Implied_Internal read of Val1, Val2, and Val3
```

This causes the final input for the three values to be Val1 × SF1, Val2 × SF1, and Val3 × SF1. The next example includes two scale factors that are applied to the values read using an *Implied_Internal*:

```
SFAC SF1
SFAC SF2
Val1 Val2 Val3
```

This causes the final input for the three values to be Val1 × SF1 × SF2, Val2 × SF1 × SF2, and Val3 × SF1 × SF2. Since the input uses an *Implied_Internal*—that is, no *Generic_Input* keyword was found so the *List Style* input is read along the current line—the previous input box can be reformatted to take advantage that an *Implied_Internal* allows specifying a single **SFAC** to the right of its input. The following illustrates this:

```
SFAC SF1
Val1 Val2 Val3  SFAC SF2
```

Note that the line of text that uses an *Implied_Internal* is preloaded into memory. By default, the maximum preloaded line size is set to 700 characters of text. Assuming there are no keywords on the line, 700 characters is approximately 35 numbers that are each 20 characters long (assuming the space that separates the numbers is part of the 20 characters). Because of this preloaded line character limit, it is not recommended to use an *Implied_Internal* when input exceeds 25 numbers. If the input must use an *Implied_Internal* with a text line that exceeds 700 characters, then the post-keyword "**DIM** DIM_SIZE" (fig. 1.12 item F) can be used to set the preloaded line to a length of DIM_SIZE characters.

When the input exceeds 25 numbers, the use of *Implied_Internal* is not recommended and instead the input should explicitly supply the *Generic_Input* keyword. The following example uses the **INTERNAL** keyword to load the three variables—the same Val1, Val2, and Val3 as presented before. As presented in figure 1.17, scale factors are always loaded before the input data, but **SFAC** may be located before or after the **INTERNAL** keyword.

If the input data reside in an external file, then the **SFAC** keyword can appear before the **EXTERNAL** or **OPEN/CLOSE** keywords or in the file that is opened. In both cases, **SFAC** must appear before the actual input data.

```
SFAC SF1
OPEN/CLOSE MyFile.txt
```

MyFile.txt is as follows:

```
ID1  Val1
ID2  Val2
ID3  Val3
```

or

```
OPEN/CLOSE  MyFile.txt
```

MyFile.txt is as follows:

```
SFAC SF2
ID1  Val1
ID2  Val2
ID3  Val3
```

or

```
SFAC SF1
OPEN/CLOSE  MyFile.txt
```

MyFile.txt is as follows:

```
SFAC SF2
ID1  Val1
ID2  Val2
ID3  Val3
```

Because MyFile.txt is opened with *Generic_Input*, it supports the optional **SF** SCALE value. The following includes a third scale factor:

```
SFAC SF1
OPEN/CLOSE  MyFile.txt  SF3
```

MyFile.txt is as follows:

```
SFAC SF2
ID1  Val1
ID2  Val2
ID3  Val3
```

This would cause the final data input to be Val1 × SF1 × SF2 × SF3, Val2 × SF1 × SF2 × SF3, and Val3 × SF1 × SF2 × SF3.

The previous **SFAC** examples relied on the ULOAD *Implied_Internal* to load the scale factors, but they can also be in an external file opened with *Generic_Input*. The following are examples that have the scale factors in a separate file:

```
SFAC OPEN/CLOSE  SFAC1.txt
OPEN/CLOSE  MyFile.txt
```

SFAC1.txt contains the following:

```
ID1  SF1
```

MyFile.txt contains the following:

```
ID1  Val1
ID2  Val2
ID3  Val3
```

The next example illustrates the use of **DIMKEY** to load more than one scale factor. **DIMKEY** is defined by the data set that supports **SFAC** and determines how the scale factors are applied when **DIMKEY** is used. An example **DIMKEY** from FMP is **ByWBS**, which indicates there are **NWBS** scale factors, and the first scale factor is applied to any input data item that is in WBS 1, and the second scale factor is applied where WBS 2 is located, and so on. For simplicity, the following example uses the **DIMKEY** "**ByVal**" and loads three scale factors that are applied to the three input values, respectively:

```
SFAC ByVal OPEN/CLOSE  SFAC3.txt
INTERNAL
ID1  Val1
ID2  Val2
ID3  Val3
```

SFAC3.txt contains the following:

```
ID1  SF1
ID2  SF2
ID3  SF3
```

This causes the final data input to be `Val1` × `SF1`, `Val2` × `SF2`, and `Val3` × `SF3`. The original `ULOAD` can contain the unscaled data and reference a separate file (SFAC3.txt) that contains a set of scale factors. This is typically used to build a parameter estimation template by keeping the unscaled input separate from the scale factor file—such as MyFile.txt, SFAC1.txt, or SFAC3.txt—that is adjusted by the parameter estimation software.

Because **`SFAC`** relies on `ULOAD` to load the scale factors, they could technically have scale factors of scale factors of scale factors, without a limit. This is not recommended, but is presented to illustrate how the two routines are interrelated:

```
SFAC OPEN/CLOSE SFAC1.txt
INTERNAL
ID1  Val1
ID2  Val2
ID3  Val3
```

SFAC1.txt contains the following:

```
SFAC OPEN/CLOSE SFAC2.txt
ID1  SF1
```

SFAC2.txt contains the following:

```
SFAC OPEN/CLOSE SFAC3.txt
ID1  SF2
```

SFAC3.txt contains the following:

```
ID1  SF3
```

This causes the final data input to be `Val1` × `SF1`× `SF2` × `SF3`, `Val2`× `SF1`× `SF2` × `SF3`, and `Val3`× `SF1`× `SF2` × `SF3`. The following is the recommended way to obtain the identical final input (note that the inclusion of **`SF`** is optional):

```
INTERNAL SF SF1 SF SF2 SF SF3
ID1  Val1
ID2  Val2
ID3  Val3
```

```
SFAC ULOAD   ← Load a scale factor with ULOAD
ULOAD        ← Load input data with ULOAD
```

Figure 1.16. General structure of a `ULOAD` input that includes an advanced scale factor, **`SFAC`**. **`SFAC`** uses `ULOAD` to read the scale factors and the `ULOAD` on the second line reads the input to which the scale factors are applied. [`ULOAD` is an abbreviation for the *Universal Loader*.]

A

```
INTERNAL   SF SF1
ID1  Val1
ID2  Val2
ID3  Val3
```

B

```
SFAC SF1
INTERNAL
ID1  Val1
ID2  Val2
ID3  Val3
```

C

```
INTERNAL
SFAC SF2
ID1  Val1
ID2  Val2
ID3  Val3
```

D

```
SFAC SF1
INTERNAL
SFAC SF2
ID1  Val1
ID2  Val2
ID3  Val3
```

Figure 1.17. Possible locations that scale factors (**SF**) and advance scale factors (**SFAC**) can be placed within input read using ULOAD with the **INTERNAL** keyword. *A*, Input that uses the *Generic_Input* scale factor **SF** SCALE to apply SF1 to input data. *B*, **SFAC** is placed before the **INTERNAL** keyword. *C*, **SFAC** is placed after the **INTERNAL** keyword, but before the input data. *D*, One **SFAC** is placed before the **INTERNAL** keyword and another **SFAC** is placed after the **INTERNAL** keyword, but before the input data. In this case the resulting scale factor is the product of SF1 and SF2. [*Universal Loader* (ULOAD) examples use *List Style* input that reads three records identified with ID1, ID2, and ID3 being integer values of the record ID.SF1, SF2, Val1, Val2, Val3 represent actual numbers that would be used as input.]

List-Array Input Structure—Spatial-Temporal Input

This section describes the *List-Array Input* (LAI). This new MF-OWHM2 input structure facilitates initial set up as well as incorporating future updates. It also includes scale factors for calibration. LAI relies on keywords that specify the frequency of reading input and its spatial style (fig. 1.18), then reads the input data with ULOAD. The input frequency is either to load the input once and reuse it for the entire simulation, keyword **STATIC**, or to load the input every stress period, keyword **TRANSIENT**. The input style is either *List Style* (keyword **LIST**) or *Array Style* (**ARRAY**). It also offers an advanced input structure called *IXJ Style* (keyword **IXJ**), in which input depends on the package input **KEYWORD** that uses it.

All *List-Array Input* structures support specification of the **TEMPORAL_KEY**, **INPUT_STYLE**, and **INPUT** keywords (fig. 1.18). If a specific input **KEYWORD** only allows for one **TEMPORAL_KEY** or **INPUT_STYLE**, however, specifying it is optional. For example, the FMP keyword, **PRECIPITATION**—which specifies the precipitation rate over the model grid—only supports the *Array Style* input.

```
PRECIPITATION   STATIC    ARRAY OPEN/CLOSE  ./Precip.txt
PRECIPITATION   STATIC          OPEN/CLOSE  ./Precip.txt
PRECIPITATION   TRANSIENT ARRAY OPEN/CLOSE  ./Precip_TFR.txt
PRECIPITATION   TRANSIENT       OPEN/CLOSE  ./Precip_TFR.txt
```

The **STATIC** keyword indicates that the input is read once and resides in the Precip.txt text file. This is an example of *Array Style* input data for a 3-by-5 (NROW by NCOL) model grid:

```
# File Precip.txt
# Precipitation rate in length translated to 3 by 5 Model Grid
0.50  0.68  0.81  0.75  0.72
0.55  0.72  0.82  0.77  0.82
0.52  0.73  0.83  0.79  0.92
```

In LAI, the **TRANSIENT** keyword indicates that Precip_TFR.txt is the *Transient File Reader* (TFR) file. The section “*Transient File Reader* and Direct Data Files” describes in detail the structure of a TFR.

If the input keyword only allows for one possible TEMPORAL_KEY and one INPUT_STYLE, then all the keywords are optional, and the input may be loaded with ULOAD directly. For example, the FMP keyword, SURFACE_ELEVATION—which specifies the initial land-surface elevation of the model grid—can only be loaded once with *Array Style*. In this case, any of the following works for loading the input from external file, DEM.txt:

```
SURFACE_ELEVATION STATIC    ARRAY OPEN/CLOSE   ./DEM.txt
SURFACE_ELEVATION STATIC          OPEN/CLOSE   ./DEM.txt
SURFACE_ELEVATION           ARRAY OPEN/CLOSE   ./DEM.txt
SURFACE_ELEVATION                 OPEN/CLOSE   ./DEM.txt
```

An example DEM.txt is as follows:

```
# DEM.txt, Land surface elevation of 3 by 5 Model Grid
0.50  0.50  0.50  0.50  0.50
0.50  0.50  0.50  0.50  0.50
0.50  0.50  0.50  0.50  0.50
```

The last line, “SURFACE_ELEVATION OPEN/CLOSE ./DEM.txt”, directly calls ULOAD because both the TEMPORAL_KEY and the INPUT_STYLE are optional.

A special TEMPORAL_KEY is CONSTANT VALUE. The contents of VALUE must be consistent with the expected input data type (that is, integer, floating point, or text). This single value is then applied to the input keyword. CONSTANT automatically picks the INPUT_STYLE that uses the least amount of memory that is allowed. That is, if the INPUT_STYLE supports both ARRAY and LIST, then CONSTANT applies the single value as if LIST were selected. Conversely, if the INPUT_STYLE only supports the ARRAY keyword, then CONSTANT applies the single value to the array. Because CONSTANT is parsed by ULOAD, which uses *Generic_Input*, the use of CONSTANT can include the post-keyword SF SCALE to multiply SCALE with VALUE. The following *List-Array Input* example is equivalent to the previous example (that loaded uniform values for land surface elevation from DEM.txt) and illustrates that CONSTANT can produce input with spatial as well as temporal uniformity:

```
SURFACE_ELEVATION STATIC    ARRAY OPEN/CLOSE ./DEM.txt
SURFACE_ELEVATION CONSTANT 0.50
```

```
KEYWORD   TEMPORAL_KEY    INPUT_STYLE    INPUT
```

where

KEYWORD	is a package input keyword (`PIK`) that initiates the *List-Array Input* utility.	
TEMPORAL_KEY	is the temporal input keyword that is set to	
	STATIC	to indicate that input data is read once with `ULOAD`.
	TRANSIENT	to indicate that input data is read every stress period with either a *Transient File Reader* (`TFR`) or a *Direct Data File* (`DDF`).
	CONSTANT `VALUE`	to indicate that input is a single value, `VALUE`, and not changed for the duration of the simulation. When using the keyword **`CONSTANT`**, it is optional to specify **`INPUT_STYLE`** and `VALUE` is considered the **`INPUT`**.
INPUT_STYLE	is the spatial input keyword that is set to	
	LIST	to use *List Style* input.
	ARRAY	to use *Array Style* input.
	IXJ	to use *IXJ Style* input.
INPUT	is the actual input data that is loaded by `ULOAD`, `TFR`, or `DDF`. The input read frequency is defined by the **`TEMPORAL_KEY`** keyword and the spatial input style used by the **`INPUT_STYLE`** keyword.	

Figure 1.18. *List-Array Input* (`LAI`) general input structure and explanation of input.

`LAI[S,T,A,L]` Input Format Meaning

To condense writing the options available for the *List-Array Input* format, a special short-hand notation is used (fig. 1.19). In most circumstances, the **`ARRAY`** keyword (A) always reads an array in the same dimension as the model grid (NROW×NCOL), and the **`LIST`** (L) keyword only reads one property (two columns, one for the ID and one for the property). If the **`LIST`** keyword loads more than one property (L-K, with K≥2), then it rarely supports the **`ARRAY`** keyword, because it cannot represent multiple properties across a single model-grid array. If the **`ARRAY`** is not the same as the model grid, then its dimension is defined by `(I,J)`, where `I` is the number of rows, and `J` is the number of columns.

`LAI[`S, T, A, L`]` or `LAI[`S, T, A, L-K`]` or `LAI[`S, T, A`(I,J)`, L`]`

where

`LAI`	indicates that input uses the *List-Array Input* structure. The contents within the brackets, `[ ]`, indicate which keywords are supported.	
S	**`STATIC`**	keyword is supported by the `LAI` input.
T	**`TRANSIENT`**	keyword is supported by the `LAI` input.
A	**`ARRAY`**	keyword is supported by the `LAI` input. *Array Style* input uses the default size, which is the model grid size (`NROW`, `NCOL`).
L	**`LIST`**	keyword is supported by the `LAI` input. *List Style* input expects a record ID and one input property. The length of the list is defined by the input that is using `LAI`.
L-K	**`LIST`**	keyword is supported by the `LAI` input. K is set to an integer, such as “L-4”, which represents the number of properties. *List Style* input expects a record ID and K input properties. The length of the list is defined by the input that is using `LAI`.
A`(I,J)`	**`ARRAY`**	keyword is supported by the `LAI` input. `I` and `J` are the number of rows and columns, respectively, that is read using *Array Style* input.

Figure 1.19. Explanation of the *List-Array Input* (`LAI`) structure variants and special short-hand notation. This notation is used with model input keywords to indicate the `LAI` features supported by each.

The Keyword **STATIC**

The keyword **STATIC** makes use of just a single ULOAD call to read the input information and then use it for the entire simulation. For example, the following example shows how to load data for the FMP keyword **ROOT_DEPTH**—which specifies NCROP crop-root depths—with the **STATIC** keyword, and ULOAD points to the input location of root depths:

```
ROOT_DEPTH STATIC ARRAY ULOAD #Array Style Input
ROOT_DEPTH STATIC LIST  ULOAD #List Style  Input
```

For the *Array Style* input, an array the same size as the model grid is loaded by ULOAD and the crop that grows in each grid cell has the root depth specified for that row and column location. Conversely, the *List Style* would read **NCROP** rows of input with the crop ID as the record ID and the next column as the root depth of the crop. Then any grid cell where the crop ID array has crop 1 receives the root depth specified in record 1, and where the crop ID array has crop 2 receives the root depth specified in record 2, until NCROP root depths have been applied.

The following is an example for a three-by-five (NROW by NCOL) model grid with three crops (NCROP = 3) that have the root depth specified in feet, but converted to meters (0.3048 m/ft), to match the model units. It also makes use of the crop ID array, which is an integer array that specifies the locations of the crops. Note that a real simulation only allows specifying the **ROOT_DEPTH** keyword once; the two versions are represented here to illustrate the difference between **ARRAY** and **LIST**.

```
# Crop ID (LOCATION) - Array Style Input of 3 by 5 Model Grid
LOCATION STATIC ARRAY INTERNAL
   1   1   2   2   2
   1   1   2   2   3
   3   3   2   3   3
#Array Style Input 3 by 5 Model Grid
ROOT_DEPTH STATIC ARRAY INTERNAL 0.3048
   1.50  1.50  0.81  0.79  0.79
   1.50  1.50  0.81  0.81  0.50
   0.50  0.50  0.81  0.50  0.50


#List Style Input
ROOT_DEPTH STATIC LIST  INTERNAL  0.3048
   1   1.50
   2   0.81
   3   0.50
```

In this example, the *Array Style* allows for crop 2 to have two different root depths (see **green bold** numbers). The root depths read as an array are mapped cell by cell using the crop ID array allowing for crop 2 to have root depths set to 0.81 and 0.79. The *List Style* input allows for more compact input but requires crop 2 to have the same root depth for the entire model array. The final root depths applied to the model grid with *List Style* are then as follows:

```
1.50  1.50  0.81  0.81  0.81
1.50  1.50  0.81  0.81  0.50
0.50  0.50  0.81  0.50  0.50
```

This would become the following after the conversion factor (0.3048) is multiplied by the input data:

```
0.45720   0.45720   0.24689   0.24689   0.24689
0.45720   0.45720   0.24689   0.24689   0.15240
0.15240   0.15240   0.24689   0.15240   0.15240
```

Note that `ULOAD` supports **`SFAC`**, by which one could specify the scale factor either before the keyword or in the external file that is opened. The following examples show three methods that can be used to apply the feet-to-meters conversion factor for the *List Style* input. Example (1), using post-keyword **`SF`** `SCALE` is this:

```
# Using Generic_Input post-keyword SF SCALE
ROOT_DEPTH STATIC LIST OPEN/CLOSE ./Root1.txt 0.3048
```

Example (2), using keyword **`SFAC`** outside the loaded text file is this:

```
# Using SFAC before Keyword
SFAC 0.3048
ROOT_DEPTH STATIC LIST OPEN/CLOSE ./Root1.txt
```

Where Root1.txt is as follows:

```
# Root Depth – List Style
   1   1.50
   2   0.81
   3   0.50
```

Example (3), using **`SFAC`** within the loaded file Root2.txt is this:

```
# Using SFAC after Keyword
ROOT_DEPTH STATIC LIST OPEN/CLOSE ./Root2.txt
```

Where Root2.txt is as follows:

```
# Scale feet to meters then
# Root Depth – List Style
SFAC 0.3048
   1   1.50
   2   0.81
   3   0.50
```

Also, additional comments can be placed anywhere:

```
# Scale feet to meters
SFAC 0.3048                   # Comment Here
# Root Depth – List Style
# Crop 1
  1   1.50    # A Comment Here
  #
  # Crop 2
  #
  2   0.81      # A Comment Here
  # Crop 3
  3   0.50      # A Comment Here
```

Although the **STATIC** keyword relies on a single ULOAD, the **TRANSIENT** keyword relies on the *Transient File Reader* to load the input for each stress period. The next section discusses this in detail.

Transient File Reader and Direct Data Files

The *Transient File Reader* (TFR) and *Direct Data File* (DDF) file types are used by the *List-Array Input* (LAI) style to load temporally varying (**TRANSIENT**) input data. The TFR can be thought of as a special spatial-temporal input format (for example, reading precipitation arrays every stress period). The TFR and DDF are both opened in the LAI structure at the **INPUT** keyword with *Generic_Input_OptKey* and must be specified as a separate file from the one that contains the LAI keywords. This precludes opening a TFR and DDF with the keywords **INTERNAL** or using an *Implied_Internal*.

The *Generic_Input_OptKey* file-opening keywords indicate if the temporal file is a TFR or DDF. Specifically, if **DATAFILE** or **DATAUNIT** open the file, then it is a DDF, whereas the rest of the keywords open a TFR.

Input from a TFR or DDF and the files that they open are preloaded, one line at a time, into memory and parsed for keywords, *List style* input, and an *Implied_Internal* input line. For most situations, this feature can be ignored by most users—since most input is loaded with a *Generic_Input* keyword—such as, **INTERNAL**, **EXTERNAL**, **OPEN/CLOSE**, **DATAUNIT**, and **DATAFILE**, which do not exceed the preloaded line size. The memory reserved for the preloaded line is 700 characters by default, which is roughly 35 numbers that are each 20 characters long. All input from a TFR or DDF—except for reading an *Array Style* array—are loaded this way. If the input fails to load a number on a line, then an error is raised signifying either not enough numbers on the line or the preloaded line size is not large enough. This can likely occur when reading *List Style* input as an *Implied_Internal*; this is because the records are all written on one line rather than one record per line. To change the maximum size of the preloaded line, the TFR or DDF that is opened with *Generic_Input_OptKey* support the post-keyword option **DIM** DIM_SIZE, which makes the preloaded line size equal to DIM_SIZE. Although it is uncommon to exceed 700 characters in a line of a file, this post-keyword is a useful add-in response to an error being raised because the input exceeds the default DIM_SIZE. Another benefit of defining DIM_SIZE is that if the input is *Array Style* or is *List Style* that never uses *Implied_Internal*, then the preloaded line size can be reduced to the largest line—excluding lines that contain the actual array—to save memory.

Transient File Reader

The *Transient File Reader* (TFR) is a pointer file that contains a ULOAD on each row that reads input per time interval it applies to. The time interval depends on the keyword that is using the ULOAD, which is typically once per stress period. Because the TFR uses ULOAD to read each temporal input, it supports all features of ULOAD, such as comments, multiple scale factors, and *Implied_Internal* (for *List Style* input with one property). Figure 1.20 presents the general format of a *Transient File Reader* file.

The next set of examples uses the previous example for root depth (FMP keyword **ROOT_DEPTH**) and changes its **TEMPORAL_KEY** to **TRANSIENT**. The set of keywords then becomes this:

```
# Generic_Input_OptKey opens TFR.
# A TFR cannot be opened with INTERNAL, DATAFILE or DATAUNIT
ROOT_DEPTH TRANSIENT ARRAY Generic_Input_OptKey # TFR reads Array Style input
ROOT_DEPTH TRANSIENT LIST  Generic_Input_OptKey # TFR reads List Style input
```

The following examples are TFR files that specify and load the input every stress period (SP) for 10 stress periods using different ULOAD keywords that open the files defined as Root1.txt and Root2.txt in the previous section (fig. 1.21). Note that although this example uses *List Style* input, the *Array Style* works in the same manner and only differs in the input data structure. Comments are included in the sample TFR file to explain how the data are loaded.

Because the file remains open after reading with the keywords **DATAFILE**, **EXTERNAL**, and **DATAUNIT**, multiple sets of input can be specified in the same file. This allows the user to work with fewer files and have the TFR control the input order and, optionally, the scale factors. Figure 1.22 is an example of a TFR that uses **DATAFILE** to cycle through three stress periods of input.

By combining **DATAFILE**, **EXTERNAL**, and **DATAUNIT** with **REWIND**, a single file could contain a repeated block of stress-period input that is cycled through. For example, root depths could vary through the year, but are typically the same depth at the same time of the year. If a model contained stress periods that align with the months of the year, then a file Root12.txt (fig. 1.23*A*) would contain 12 sets of input for each month's root depth. The structure of the input depends on the input style, where *List Style* input expects a root depth for each crop and array style reads a model grid array that is the root depth for each model row and column. Figure 1.23*A* presents the root depth raw input using *List Style* for three crops. This raw input is then cycled through using the **DATAFILE** directive (fig. 1.23*B*). Another method of cycling through a file is using the **LOAD_NEXT** and **RELOAD** directives (fig. 1.23*C*).

Using an annual file that is cycled through each year provides an easy method for maintaining input datasets in a compact manner that is easy to understand. For more information on cycling through a TFR file, please see the "*Transient File Reader* – Spatial-Temporal Input" in appendix 2.

If one file is cycled through for the entire simulation by using only **DATAFILE**, **DATAUNIT**, or **EXTERNAL** (that is, no **REWIND**), then the *Transient File Reader* can be bypassed and loaded as a *Direct Data File* (DDF)—discussed in the next section.

```
ULOAD   # First     Load of Input
ULOAD   # Second    Load of Input
ULOAD   # Third     Load of Input
  ⋮
ULOAD   # Nth       Load of Input
```

Figure 1.20. General input structure of a *Transient File Reader* (TFR) file. If the TFR input occurs once per stress period, then this example represents reading the input for the first N stress periods. Each uncommented, non-blank line in the file is expected to load the input needed for that stress period. Input for each stress period is read with the *Universal Loader* (ULOAD). That is, the first ULOAD reads input for stress period 1, and the second ULOAD reads input for stress period 2, and so forth. [Comments are any text that are written to the write of a "#" symbol.]

```
# Transient File Reader Example that loads 10 Stress Periods (SP)
INTERNAL 0.3048 # SP1
   1   1.50
   2   0.81
   3   0.50

REPEAT  SF 0.3048     # SP2: Reuse previous input, multiply it by 0.3048
                      # Note the keyword “SF” is optional, so “REPEAT  0.3048” works.
#
# SP3: Implied Internal - SFAC must be used instead of “SF SCALE” or “SCALE”
#
SFAC 0.3048
1.50  0.81  0.50
#
# SP4:  Open and then close Root2.txt and multiply its input by 0.3048
OPEN/CLOSE ./Root2.txt   SF 0.3048
#
EXTERNAL 55     # SP5: Unit 55 is Root2.txt that is open in the NAME file
#
# Note that keywords DATAFILE and DATAUNIT work within the TFR
# Root2.txt remains open for the rest of the simulation and is buffered into 64 kb of RAM
#
DATAFILE ./Root2.txt  SF 0.3048 BUFFER 64 # SP6: Load from first line of Root2.txt
#
DATAFILE ./Root2.txt  SF 0.3048   # SP7:  Load input from current line of Root2.txt
                                  #       The current line is the next text file line
                                  #       from where the SP6 input load ended.

# SP8:  Move to first line of Root2.txt, load input, and apply SCALE (multiply by 0.3048)
RELOAD  0.3048        # Note the use of “SF” is optional

# Unit 55 is Root2.txt, but it was read already once
# so a second call to the unit with EXTERNAL or DATAUNIT
# would cause an end of file error, so the open file must be rewound back to the first line.
#
EXTERNAL  55 REWIND # SP9 move to line 1 of file, then load input
DATAUNIT  55 REWIND # SP10 Same effect as EXTERNAL 55
```

Figure 1.21. Example *Transient File Reader* (TFR) file that loads input for 10 stress periods (SP) using various TFR directives. Input is assumed to be *List Style* that reads 3 records. Comments are used to explain what each directive is doing and what is being read. [Comments are any text that are written to the right of a "#" symbol.]

A

```
# Transient File Reader file that reads from Root2.txt input for three stress periods
# Each stress period's input is scaled by 0.3048
# (Note that "SF 0.3048" could be used in the place of "0.3048")
#
DATAFILE ./Root2.txt 0.3048 # SP1: open Root2.txt, load input from first line
DATAFILE ./Root2.txt 0.3048 # SP2: Root2.txt already open, load at current file location
DATAFILE ./Root2.txt 0.3048 # SP3: continue loading input from Root2.txt
```

B

```
# Root2.txt
   1   1.50   # SP 1 Root Depth – List Style
   2   0.81
   3   0.50
# SP 2          -- Note that comment does not have to be on separate line
   1   1.50
   2   0.81
   3   0.50
# SP 3
   1   1.50
   2   0.81
   3   0.50
```

Figure 1.22. Example *Transient File Reader* (TFR) file that loads input for three stress periods (SP) using the DATAFILE directive and *List Style* input that expects 3 records. Comments are used to explain what each directive is doing and what is being read. *A*, The TFR file that opens and reads input from Root2.txt once per SP. *B*, The input file Root2.txt that is read from. In the file, the comments are optional and used to separate, visually, each stress period's input. [Comments are any text that are written to the right of a "#" symbol.]

A

```
# Root12.txt
#  ID    Root_Depth
    1    0.20   #SP -- January
    2    0.21
    3    0.10
    1    0.60   #SP -- February
    2    0.71
    3    0.30
    1    1.50   #SP -- March
    2    0.81
    3    0.50

    1    1.50   #SP -- November
    2    0.81
    3    0.50
    1    0.10   #SP -- December
    2    0.15
    3    0.05
```

B

```
DATAFILE  ./Root12.txt  0.3048          # SP1    January
DATAFILE  ./Root12.txt  0.3048          # SP2    February
DATAFILE  ./Root12.txt  0.3048          # SP3    March
DATAFILE  ./Root12.txt  0.3048          # SP4    April
DATAFILE  ./Root12.txt  0.3048          # SP5    May
DATAFILE  ./Root12.txt  0.3048          # SP6    June
DATAFILE  ./Root12.txt  0.3048          # SP7    July
DATAFILE  ./Root12.txt  0.3048          # SP8    August
DATAFILE  ./Root12.txt  0.3048          # SP9    September
DATAFILE  ./Root12.txt  0.3048          # SP10   October
DATAFILE  ./Root12.txt  0.3048          # SP11   November
DATAFILE  ./Root12.txt  0.3048          # SP12   December
DATAFILE  ./Root12.txt  0.3048  REWIND  # SP13   January - Rewind to top of file
DATAFILE  ./Root12.txt  0.3048          # SP14   February
DATAFILE  ./Root12.txt  0.3048          # SP15   March
```

Figure 1.23. Example *Transient File Reader* (TFR) file that cycles through a file that contains 12 months of root depth input for three crops (*List Style* input with three records). In this example, the TFR time interval is a Stress Period (SP) defined equivalently with the months of the year. For example, SP 1 and 13 represent January and SP 2 and 14 represent February. *A*, The raw input file, Root12.txt, which contains the root depth for three crops for each month of the year. If a model only used between 1 and 12 Stress Periods, then Root12.txt could also be used opened as a *Direct Data File* (DDF). *B*, An example TFR file that uses the **DATAFILE** directive to open and read input from Root12.txt for each SP. *C*, An example TFR file that uses the **DATAFILE** directive to open and read input from Root12.txt initially and then uses the **LOAD_NEXT** and **RELOAD** directives to cycle through the file for each SP. [Comments are any text that are written to the right of a "#" symbol. The TFR examples use a scale factor, SCALE = 0.3048 that multiplies with the root depth to convert the depths from feet to meters.]

C

```
DATAFILE  ./Root12.txt  0.3048          # SP1    January
LOAD_NEXT   0.3048                      # SP2    February
LOAD_NEXT   0.3048                      # SP3    March
LOAD_NEXT   0.3048                      # SP4    April
LOAD_NEXT   0.3048                      # SP5    May
LOAD_NEXT   0.3048                      # SP6    June
LOAD_NEXT   0.3048                      # SP7    July
LOAD_NEXT   0.3048                      # SP8    August
LOAD_NEXT   0.3048                      # SP9    September
LOAD_NEXT   0.3048                      # SP10   October
LOAD_NEXT   0.3048                      # SP11   November
LOAD_NEXT   0.3048                      # SP12   December
RELOAD      0.3048                      # SP13   January - Rewind to top of file
LOAD_NEXT   0.3048                      # SP14   February
LOAD_NEXT   0.3048                      # SP15   March
```

Figure 1.23. —Continued

Direct Data File

The *Direct Data File* (DDF) is opened when the LAI uses the *Generic_Input* keyword **DATAFILE** or **DATAUNIT**. Functionally, the DDF acts identically to a *Transient File Reader* (TFR) file that only contains **DATAFILE**, **DATAUNIT**, or **EXTERNAL** references with no **SFAC** or post-keywords (for example, **SF SCALE**, **REWIND**). The DDF allows a shortcut for accessing an input file without having to double-specify keywords. A DDF is the closest analogue to how input was loaded by previous versions of FMP and can help with transforming previous FMP inputs to the current input design. To translate such an older structure to a LAI structure, the same **EXTERNAL** file could be opened with **DATAUNIT** as with the LAI **INPUT** keyword.

The following example presents a TFR and its equivalent DDF using the previously discussed root depth examples with *List Style* input for three crops and for four stress periods; the base text file Root3.txt is shown first:

```
# Root3.txt – Stress period input is defined as contiguous blocks – no keywords
   1   1.0        # SP 1 Root Depth – List Style
   2   0.8
   3   0.5
   1   1.1        # SP  2
   2   0.9
   3   0.6
   1   1.2        # SP 3
   2   0.95
   3   0.7
   1   1.3        # SP 4
   2   0.99
   3   0.8
```

Next, the FMP input keyword `ROOT_DEPTH` uses keywords to indicate *List-Style* input lines will be loaded using the *Transient File Reader* (`TFR`) file, Root_TFR.txt, opened as is shown:

```
# Generic_Input_OptKey opens TFR.
# A TFR cannot be opened with INTERNAL, DATAFILE or DATAUNIT
ROOT_DEPTH TRANSIENT LIST OPEN/CLOSE ./Root_TFR.txt # Reads List Style input
```

```
# Root_TFR.txt
DATAFILE ./Root3.txt 0.3048    # SP1, Open Root3.txt, read from first line
DATAFILE ./Root3.txt 0.3048    # SP2, continue reading from Root3.txt
DATAFILE ./Root3.txt 0.3048    # SP3, continue reading from Root3.txt
DATAFILE ./Root3.txt 0.3048    # SP4, continue reading from Root3.txt
```

Alternatively, Root_TFR.txt could use **`EXTERNAL`** by declaring Root3.txt in the NAME file:

```
# Root_TFR.txt, Unit 56 is Root3.txt opened in the NAME file
EXTERNAL 56  0.3048    # SP1, read from unit 56 (Root3.txt)
EXTERNAL 56  0.3048    # SP2, read from unit 56 (Root3.txt)
EXTERNAL 56  0.3048    # SP3, read from unit 56 (Root3.txt)
EXTERNAL 56  0.3048    # SP4, read from unit 56 (Root3.txt)
```

This `LAI` structure using a `TFR` could be translated into one that opens a `DDF` by using the keyword **`DATAFILE`** or **`DATAUNIT`** as part of the `LAI` **`INPUT`**. The following example directly accesses Root4.txt and bypasses the `TFR`:

```
# Direct Data File:  Is only opened with DATAFILE or DATAUNIT
#
ROOT_DEPTH TRANSIENT LIST DATAFILE ./Root3.txt 0.3048 # Open as DDF
#
# or if Root3.txt is opened in the NAME file on Unit 56
#
ROOT_DEPTH TRANSIENT LIST DATAUNIT 56 0.3048
```

The limitation of the `DDF` is it does not allow for easy specification of time-varying scale factors. The only way a scale factor can vary in a `DDF` is by specifying it with the **`SFAC`** keyword before each stress-period input. The following example file illustrates the only method for applying a temporally varying scale factor when input is opened as a `DDF`. This example reads the input for four stress periods and applies the scale factors 1.1, 1.2, 1.3, and 1.4, to stress periods 1, 2, 3, and 4, respectively.

```
SFAC 1.1          # SFAC applied to first read of DDF
   1   1.0        # SP 1 Root Depth – List Style
   2   0.8
   3   0.5
SFAC 1.2          # SFAC applied to second read of DDF
   1   1.1        # SP 2
   2   0.9
   3   0.6
SFAC 1.3
   1   1.2        # SP 3
   2   0.95
   3   0.7
SFAC 1.4
   1   1.3        # SP 4
   2   0.99
   3   0.8
```

IXJ Style Input—Advanced Structured Input

The *IXJ Style* input is an advanced input that serves as a surrogate to the *List* and *Array Style* inputs. The *IXJ Style* has been intentionally left out of the previous descriptions to prevent confusion with the recommended input structures (**`LIST`** and **`ARRAY`**). It is not necessary to use the *IXJ Style* input structure as part of the standard *List-Array Input*. It is documented here for completeness and to provide examples of the benefits of using it. *IXJ Style* input is loaded using `ULOAD`, so the *IXJ Style* supports all of the `ULOAD` keywords and scale factors (**`SFAC`**). The name " `IXJ` " is a reference to the loading of an arbitrary number of lines of input in which each line has a prespecified set of integers (`I`), followed by a set of floating-point numbers (`X`), and finally by a second set of integers (`J`). The number of integers (`I` and `J`) and floating-point numbers (`X`) that are read on each line depends on the input keyword that loads them. The loading of *IXJ Style* input continues until it either reaches the end of the file or encounters the keyword **`STOP IXJ`**.

The dimensions `I`, `X`, and `J` for *IXJ Style* input are either explicitly defined or specified with a shorthand notation. The shorthand notation has the structure of `IXJ[`$\texttt{DIM}_\texttt{I}$, $\texttt{DIM}_\texttt{X}$, $\texttt{DIM}_\texttt{J}$`]`, where $\texttt{DIM}_\texttt{I}$, $\texttt{DIM}_\texttt{X}$, $\texttt{DIM}_\texttt{J}$ are set to a number that represents the number of integers read for `I`, floating-point real numbers read for `X`, and integers read for `J`, respectively. For example, `IXJ[3, 1, 0]` indicates that the *IXJ Style* input expects to read on each row three integers (`I`), one floating-point (`X`), and zero integers (`J`). The shorthand can optionally exclude $\texttt{DIM}_\texttt{J}$, which indicates its value is zero. Using this option would shorten the previous example to `IXJ[3, 1]`.

A common use of the *IXJ Style* is to specify *Array Style* using a compressed coordinate structure in which each line contains the row number, the column number, and the floating-point value to assign at that row and column location. Any row and column location that is not defined in the *IXJ Style* is set to zero (in fact, the array is initialized to zero, and then the *IXJ Style* input overwrites each specified row and column location with the assigned value). Using *IXJ Style*, the compressed coordinate structure would read two values for `I`, one value for `X`, and zero values for `J`.

As an example of input using the *IXJ Style* as surrogate for *Array Style*, reconsider the previously used **`ARRAY`** input for **`PRECIPITATION`** (shown in the example following fig. 1.18). The following is an example use of `LAI` for precipitation input using *Array Style* where Precip.txt (fig. 1.24*E*) contains the *Array Style* precipitation data:

```
PRECIPITATION   STATIC    ARRAY OPEN/CLOSE   ./Precip.txt
```

The same data can be loaded as *IXJ Style* input by translating the precipitation array to *IXJ Style* (fig. 1.24*B*) and changing the input style keyword from **`ARRAY`** to **`IXJ`**. This would change the previous example as follows:

```
PRECIPITATION   STATIC    IXJ OPEN/CLOSE   ./PrecipIXJ.txt
```

Figure 1.24 presents a precipitation *List Array Input* example that uses the **`TRANSIENT`** keyword and *IXJ Style* input (fig. 1.24*A*). Figure 1.24*B* is the `TFR` that reads two stress periods; the first stress period uses **`INTERNAL`** directive to load five lines of *IXJ Style* input, and the second stress period uses **`OPEN/CLOSE`** directive to read the *IXJ Style* input from the file PrecipIXJ.txt (fig. 1.24*C*). If a row and column is not specified, then *IXJ Style* automatically assumes a value of zero for precipitation. The final precipitation arrays for the two stress periods are presented in figure 1.24*D* and *E*.

The *IXJ Style* is suited best for input of the land-use area fractions if there are multiple land-use types allowed in one model cell. This advanced feature in the FMP allows more than one land use type for each model cell. The input keyword that specifies crop-area fractions is **`LAND_USE_AREA_FRACTION`**, and it expects *Array Style* input that reads multiple `NROW` by `NCOL` arrays (one per Crop) of fractions of the cell area. Figure 1.25*A* is an example that loads such arrays for a three-by-five (`NROW` by `NCOL`) model grid's crop-area fractions for three crops (`NCROP = 3`). The equivalent *IXJ Style* input (fig. 1.25*B*) reads three integers and one floating-point number. The three integers are crop ID, row, and column, and the floating-point number is the fractional cell area applied to the specified crop ID at that model grid row and column. The advantage of using *IXJ Style* is that only the non-zero fractions must be specified. In this example, the difference between the *IXJ* and *Array Styles* is trivial, but the advantage can be quite substantial for large scale models.

A

```
PRECIPITATION   TRANSIENT IXJ OPEN/CLOSE ./PrecipIXJ_TFR.txt
```

B

```
# PrecipIXJ_TFR.txt
INTERNAL                   # SP1
#ROW  COLUMN  PRECIP
 1    3       0.81
 2    1       0.55
 2    2       0.72
 2    4       0.77
 3    4       0.79
STOP IXJ                   # Keyword that ends loading IXJ input for stress period
#
OPEN/CLOSE PrecipIXJ.txt   # SP2 loads from file PrecipIXJ.txt
```

C

```
# File: PrecipIXJ.txt
#ROW  COLUMN  PRECIP
 1    1       0.50
 1    2       0.68
 1    3       0.81
 1    4       0.75
 1    5       0.72
 2    1       0.55
 2    2       0.72
 2    3       0.82
 2    4       0.77
 2    5       0.82
 3    1       0.52
 3    2       0.73
 3    3       0.83
 3    4       0.79
 3    5       0.92
#
# End of file indicates termination of input;
# or could use keyword
# STOP IXJ
```

Figure 1.24. *A*, Example package input keyword, **PRECIPITATION**, that uses *List-Array Input* with IXJ Style to specify for two stress periods the precipitation rate for a 3 by 5 model grid. *B*, Example *Transient File Reader* file, PrecipIXJ_TFR.txt. The first stress period is loaded with the **INTERNAL** directive and reads on each uncommented, non-blank line, the row and column number of the model grid and the precipitation rate that is assigned to it. *IXJ Style* input continues to read lines until encountering the keyword “STOP IXJ”. The second stress period loads the *IXJ Style* input from the file PrecipIXJ.txt. *C*, The file PrecipIXJ.txt, that contains *IXJ Style* input, is read until the end of file is reached. *D*, The resulting array that MF-OWHM2 uses after reading the *IXJ Style* input specified by the **INTERNAL**. *E*, The resulting array that MF-OWHM2 uses after reading PrecipIXJ.txt with *IXJ Style* input. [Comments are any text that are written to the right of a “#” symbol.]

D

```
# Resulting Precipitation array from IXJ Style read with INTERNAL
# Precipitation rate in length translated to 3 by 5 Model Grid
0.00  0.00  0.81  0.00  0.00
0.55  0.72  0.00  0.77  0.00
0.00  0.00  0.00  0.79  0.00
```

E

```
# File Precip.txt
# Resulting Precipitation array from IXJ Style read of PrecipIXJ.txt
# Precipitation rate in length translated to 3 by 5 Model Grid
0.50  0.68  0.81  0.75  0.72
0.55  0.72  0.82  0.77  0.82
0.52  0.73  0.83  0.79  0.92
```

Figure 1.24. —Continued

A

```
#Array Style Input 3 by 5 Model Grid and NCROP=3
LAND_USE_AREA_FRACTION STATIC ARRAY INTERNAL
#                       CROP 1
0.5  0.0  0.0  0.0  0.0
0.2  0.0  0.0  0.0  0.2
0.0  0.0  0.3  0.0  0.0
#                       CROP 2
0.0  0.8  0.0  0.0  0.0
0.8  0.8  0.0  0.0  0.0
0.0  0.0  0.3  0.0  0.0
#                       CROP 3
0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.5  0.5
0.0  0.0  0.3  0.5  1.0
```

B

```
#IXJ Style Input 3 by 5 Model Grid and NCROP=3
LAND_USE_AREA_FRACTION STATIC IXJ INTERNAL
#Crop  Row  Col   Fraction
  1     1    1     0.5
  1     2    1     0.2
  1     2    5     0.2
  1     3    3     0.3
  2     1    2     0.8
  2     2    1     0.8
  2     2    2     0.8
  2     3    3     0.3
  3     2    4     0.5
  3     2    5     0.5
  3     3    3     0.3
  3     3    4     0.5
  3     3    5     1.0
STOP IXJ
```

Figure 1.25. Examples of *List-Array Input* for the FMP Land_Use block using input keyword `LAND_USE_AREA_FRACTION` to *A*, load an *Array Style* input for 3 crops and a model grid that is 3 rows by 5 columns. This particular input expects to read an array that is 9 rows (3 sets of 3) by 5 columns, where the first 3 by 5 array represents the fraction of each cell that crop 1 is planted, and the second 3 by 5 array is for crop 2, and so forth; and *B*, loads an *IXJ Style* input equivalent to the *List-Array Input* of part *A*. [Comments are any text that are written to the right of a "#" symbol. Comments are optional and included to help organize the example.]

Lookup Table Input Structure

The Lookup Table input structure reads *Lookup Style* input; that is, it uses lookup tables that are composed of pairs of data—the lookup value and its associated return value. In all lookup tables, the lookup values should be sorted in ascending order, but if they are not, then MF-OWHM2 sorts the lookup table rows to ensure ascending order. An individual lookup table is read once (for example, a single stream stage-discharge table), but may be used for different lookup values or during different times of a simulation (such as evaluating the discharge rate given a stream stage at the start of each stress period). The *Lookup Style* input within a `LAI` currently only supports the **`STATIC`** **`TEMPORAL_KEY`** keyword—that is, `LAI` reads all the lookup tables once and reuses them for the entire simulation.

In MF-OWHM2, a lookup table is optimized for fast searches within small tables—less than 1,000 rows of data—with basic interpolation methods. There is no limit to the size of a lookup table, but it is recommended to resample the table to make it smaller if it contains redundant or unnecessary information.

A lookup value may be specified as a date (using any of the date formats described in appendix 2), but MF-OWHM2 internally converts each value in the lookup column to an equivalent decimal year that is treated as a regular floating-point number, and the specified lookup value also is converted likewise for comparison. For example, if the lookup value is specified as 1979-4-23, 1979-4, or APR-23, these are converted to `1979.307`, `1979.247`, and `0.307`, respectively. Note that if the day of month is not specified it is assumed to be 1. If the year is not specified, then it is assumed to be zero. Figure 1.26 is an example lookup table composed of nine rows.

There are four methods available for determining the return value for a given search value to seek among the lookup values. The method is set at the time of loading the lookup table and cannot change during a simulation. The first method, signified by **`METHOD`** keyword **`NEAREST`**, searches for closest lookup value and returns the paired return value. If the search value is equidistant from two lookup values, then the larger lookup value is selected. The second method, signified by **`METHOD`** keyword **`STEP_FUNCTION`**, is a step function that searches for the closest lookup value that is less than or equal to the search value. For the **`STEP_FUNCTION`** method, if the search value is less than the first lookup value then the first return value is used. The third option, signified by **`METHOD`** keyword **`INTERPOLATE`**, linearly interpolates a return value based on the lookup values bracketing the search value. If the search value is less than the smallest lookup value in the table then the linear interpolation uses the first two rows of values to extrapolate a return value. Similarly, if the search value exceeds the table's largest lookup value, then the last two rows are used to extrapolate. The fourth method, signified by **`METHOD`** keyword **`CONSTANT`**, is a special case that forces the table to always return the same value. Figure 1.27 presents an example Lookup Table and the results from three different methods for given search values.

A set of lookup tables can be loaded with `ULOAD`, which allows lookup tables to be loaded with `LAI`; however, limitations exist—only the `LAI` keyword **`LIST`** is supported for lookup tables. Specifically, MF-OWHM2 only supports *List Style* input that reads one lookup table for each record. The Lookup Table input structure (*Lookup Style* input, fig. 1.28) is composed of three parts, the first being the **`METHOD`** keyword that indicates how the lookup table returns a value—**`NEAREST`**, **`STEP_FUNCTION`**, **`INTERPOLATE`**, or **`CONSTANT`**. The second part, `NTERM`, is the number of rows in the lookup table, and the last part is a *Generic_Input* file identifier that points to where the lookup table is. A scale factor, **`SF`** `SCALE`, maybe optionally specified after the *Generic_Input*. At the start of the *Generic_Input* file the keyword **`SFAC`** is supported, but it does not support any **`DIMKEY`** keywords—that is, it only supports loading a single scale factor. If a scale factor is read, then it is only applied to the return value of the lookup table (the second column). Figure 1.28 is a formal description of *Lookup Style* input.

Figure 1.29 is an example package input **`KEYWORD`** that uses *List Style* input to read a list of three record ID's that use *Lookup Style* input. The lookup table associated with record ID 1 is loaded from the file TAB.txt and specifies that it will find return values using the **`NEAREST`** method. Because `NTERM` is set to zero, the number of rows in the table is automatically determined during reading of TAB.txt. The table associated with record ID 2 will use the **`INTERPOLATION`** lookup method and contains 4 rows. The **`INTERNAL`** keyword indicates that the lookup table is loaded on the subsequent lines. The lookup table associated with record ID 3 has the lookup method set to **`CONSTANT`**, so it will always return 0.5 for all search values.

```
# LookUp  ReturnValue
  1.0     19.0
  2.0     66.0
  5.0    -91.0
 10.0    -82.0
 12.0     17.0
 12.4    -57.0
100.0    -75.0
123.0     23.0
956.0     91.0
```

Figure 1.26. Example lookup table used by the Lookup Table input structure (*Lookup Style*). The first column of numbers are the lookup values and the second column of numbers are the associated return values.

A

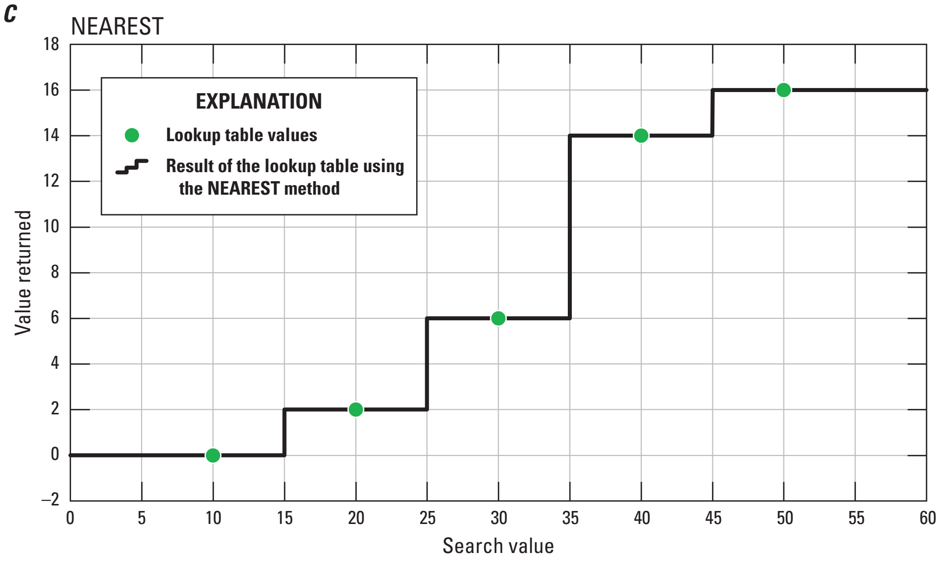
```
# LookUp  ReturnValue
   10.0      0.0
   20.0      2.0
   30.0      6.0
   40.0     14.0
   50.0     16.0
```

B

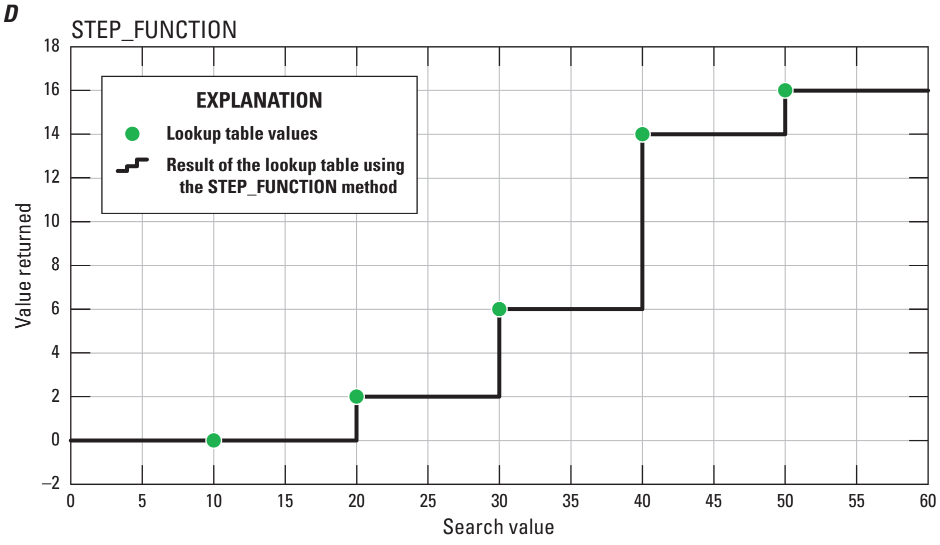
	Value Returned by METHOD		
Search Value	NEAREST	STEP_FUNCTION	INTERPOLATE
5	0	0	-1.0
14	0	0	0.8
15	2	0	1.0
16	2	0	1.2
29	6	2	5.6
30	6	6	6.0
31	6	6	6.8
55	16	16	17.0

Figure 1.27. Summary of Lookup Table input (*Lookup Style*) examples for various search values given three of the available lookup methods: *A*, example lookup table; *B*, return values obtained by each method for the given set of search values; graphs showing lookup table's returned values using the *C*, **NEAREST** method, *D*, **STEP_FUNCTION** method, and *E*, **INTERPOLATE** method.

C



D



E

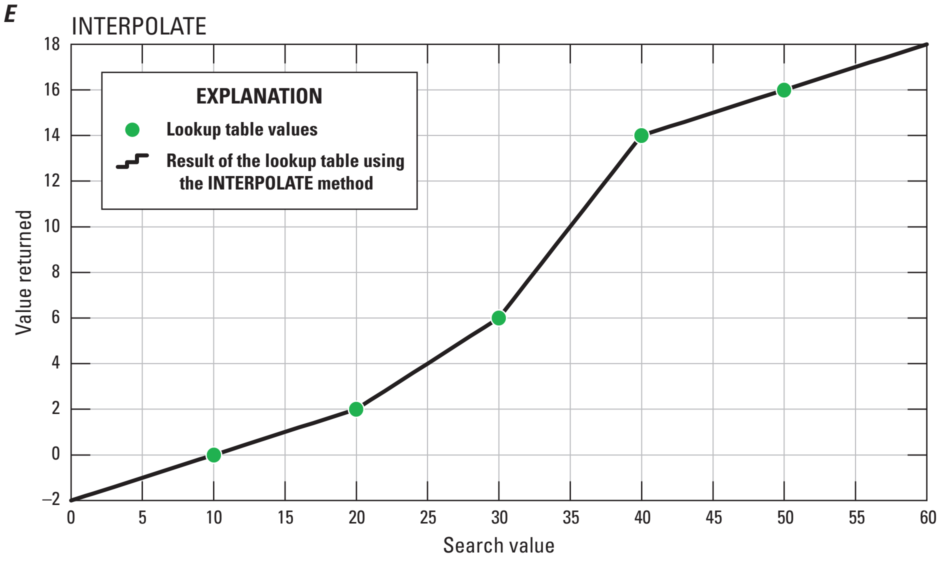


Figure 1.27. —Continued

`METHOD NTERM` *Generic_Input_OptKey*

where

`METHOD`	defines the method that is used for determining the return value based on the relationship between the search value and the lookup value. It must be set to one of the following keywords:

`NEAREST`	the nearest lookup value determines the return value.
`STEP_FUNCTION`	the nearest smaller or equal lookup value determines the return value.
`INTERPOLATE`	linearly interpolates between the lookup values to the search value to determine the return value.
`CONSTANT` `VALUE`	declares that lookup table is composed of a single number that is returned for all lookup values. `VALUE` must be specified after the keyword **`CONSTANT`** and is the number that will be returned. Because they are not needed, `NTERM` and *Generic_Input* are not read.

`NTERM`	is the number of rows in the lookup table. If set to a negative value or zero, then the lookup table must reside in a separate file and its size is automatically determined. The length is determined based on the number of successfully loaded, uncommented, rows in the table (namely the bottom of the file is reached or there is a non-commented line that fails to load).
Generic_Input_OptKey	is the input file that contains the lookup table. If there is a scale factor, `SCALE`, specified then it is only applied to the return values (second column).

Figure 1.28. General input structure for *Lookup Style* input, which loads a single lookup table and specifies the lookup method by which return values will be associated with a search value.

A

```
#
KEYWORD STATIC LIST INTERNAL
#ID Lookup-Style
#ID METHOD   NTERM GENERIC_INPUT
1  NEAREST     0     TAB.txt   # Load table in TAB.txt; auto-count rows
2  INTERPOLATE 4    INTERNAL   # Table with 4 rows on subsequent lines
                    10   0
                    20   2
                    30   8
                    40  14
3  CONSTANT    0.5             # Table always returns 0.5
```

B

```
# File: TAB.txt
# LookUp   ReturnValue
  5.0       25.0     # First Value
  8.0       50.0
 10.0       75.0
         # Comments can be anywhere
 15.0       80.0
 50.0       98.0
# End of file determines table size is 5 = NTERM
```

Figure 1.29. Example using *Lookup Style* input. The shorthand notation is **KEYWORD** LAI[S, L] using *Lookup Style*, where **KEYWORD** represents the package input keyword that supports lookup tables. The *List Style* input reads three lookup tables (three records) that each define the lookup method and lookup table location. *A*, The input for **KEYWORD** LAI[S, L] using *Lookup Style*. *B*, The lookup table specified in part *A* as the TAB.txt file.

References Cited

Harbaugh, A.W., 2005, MODFLOW-2005—The U.S. Geological Survey modular ground-water model—The ground-water flow process: U.S. Geological Survey Techniques and Methods 6–A16, variously paginated, https://pubs.usgs.gov/tm/2005/tm6A16/.