Theory for WORK 7 onwards

Using logical units (files)

Input (reads) and output (writes) instructions are performed through logical units. The default logical unit (marked with the value `*` in statements that require specification) is the console, i.e. the assembly consisting of the keyboard and display (monitor screen) – for inputs the keyboard is considered, and for outputs the display. The logical units that must be explicitly specified are files, respectively peripherals (printer, magnetic tape drive, etc.), with an integer value assigned to them. The indication of the logical unit to which an input / output instruction refers is done by this numerical value. Some values also have predefined logical units in older Fortran, such as 1, 2, 3 and 4 for files named `FOR00`*n*`.DAT` (where the corresponding digit from 1 to 4 will appear instead of the character *n* in the file name), or 5 for input devices (card reader, keyboard, etc.) and 6 for output devices (printer, display, etc.). Allocating these references (numbers) to logical units can be done explicitly by the `OPEN` statement. The syntax of this executable statement is as follows:

`OPEN(`*parameter*`[,`*parameter*`]…)`

where *parameter* can be a *keyword*, or of the form *keyword*=*value* (each parameter may be specified only once within the list in parentheses).

Table of parameters in the `OPEN` statement – the blue ones are not accepted by the G95 compiler (in alphabetical order, without the Intel Fortran QuickWIN specific ones):

keyword	*value*	Explanations	Implicit value

`ACCESS=`	`"SEQUENTIAL"`
	`"DIRECT"`
	`"APPEND"`
	`"KEYED"`
		Setting how to access the logical unit:
		- sequential,
		- direct,
		- addition,
		- using key-fields.
			`"SEQUENTIAL"` (row by row).

`ACTION=`
`MODE=`
	`"READ"`
	`"WRITE"`
	`"READWRITE"`
		How to use the logical unit:
		- only to read from it,
		- only to write in it,
		- reading and writing.
			`"READWRITE"` (read and write)

`ASYNCHRONOUS=`	`"NO"` `"YES"`	Allows specification of asynchronous I/O mode.	`"NO"` (synchronous I/O operations).
`ASSOCIATEVARIABLE=`	*number*	The number of the next record in case of direct access (*number* being a positive integer).	There is no implicit value.
`BLANK=`	`"NULL"` `"ZERO"`	Interpretation of blanks: - spaces (no conversion), - `0` (conversion to digits for numbers).	`"NULL"` (no conversion).
`BLOCKSIZE=`	*number*	Size of a block in the I/O buffer (*number* being a positive integer).	Value set by the operating system.
`BUFFERCOUNT=`	*number*	Number of input/output buffers (*number* being a positive integer).	Value set by the operating system.
`BUFFERED=`	`"NO"` `"YES"`	It allows specifying the behavior of run-time libraries after write operations performed on the logical unit:	`"NO"` (directly without using a supplemental buffer). Caution, the operating system

		- no additional buffer used, - use of additional buffer memory.	will use buffer memory for write operations!
CARRIAGECONTROL=	"FORTRAN" "LIST" "NONE"	Controlling (interpreting) the carriage return (the character generated when the <*Enter*> key is pressed): - Fortran (the first character will be interpreted and "consumed"), - list (the carriage return is the last character), - none.	"FORTRAN" in the case of the "FORMATTED" form, and "NONE" in the case of the "UNFORMATTED" form.
CONVERT=	"NATIVE" "SWAP" "LITTLE_ENDIAN" "BIG_ENDIAN" "CRAY" "FDX" "FGX" "IBM" "VAXD" "VAXG"	Allows specifying a numeric format (for conversion / interpretation) for unformatted data: - native (no conversion), - switch (between LITTLE_ENDIAN and BIG_ENDIAN), - various other formats…	"NATIVE" (no conversion).
DEFAULTFILE=	*expresie_caracter*	Setting a default file specification.	None.
DELIM=	"NONE" "APOSTROPHE" "QUOTE"	Specifying the delimiter character (for CHARACTER type constants) for I/O operations: - without delimiter, - the apostrophe character, - the quotation mark character.	"NONE" (no delimiter).
DISPOSE= DISP=	"SAVE" "KEEP" "PRINT" "DELETE"	The state of the logical unit (usually the file) when closing: - save, - keep (temporary), - print, - delete.	"SAVE" (the content is saved).
ERR=	*label*	Instruction *label* to jump to in case of error when opening the logical unit.	Not implicit, no jump by default.
EXTENDSIZE=	*number*	Size of the storage space allocated for the file (*number* being a positive integer).	Given by the operating system or by the volume (partition).
FILE= NAME=	*character_string*	Specify the file to be used as the logical unit. The file specifier is considered a string, so it is delimited by apostrophe or quotation marks if quoted, and if contained by aCHARACTER,	It depends on the logical unit and the operating system.

		entity, the name of that entity may be specified.	
FORM=	"FORMATTED" "UNFORMATTED" "BINARY"	Format of the logical unit (file) accessed: - formatted, - unformatted, - binary.	Depends on the value of the ACCESS keyword. If it is "DIRECT" or "KEYED" then it will be considered "FORMATTED", otherwise it will be considered "UNFORMATTED".
INITIALSIZE=	*number*	The initial memory size allocated to the file (*number* being a positive integer).	Not allocated
IOSTAT=	*variable*	Returns a scalar INTEGER value in the variable, indicating the success (or failure) of accessing the logical drive. If the logical unit is opened successfully, the value of the variable is 0.	Not implicit.
KEY=	(*key1*[, *key2*]...)	The key fields (in order of their priority) for the indexed file.	Not implicit.
MAXREC=	*number*	The maximum number of records that can be transferred in direct access (*number* being a positive integer).	No maximum.
NOSPANBLOCKS		Records do not span over memory blocks.	Records may span over memory blocks.
ORGANIZATION=	"SEQUENTIAL" "RELATIVE" "INDEXED"	Structure (organization) of the logical unit (file): - sequential, - relative, - indexed.	"SEQUENTIAL".
PAD=	"YES" "NO"	Specifies whether a record is filled with spaces (blank characters) when the format requires more positions than the value entered, or not filled.	"YES" (blanks are used when necessary).
POSITION=	"ASIS" "REWIND" "APPEND"	Specifies the positioning in a file: - as is, - back to the beginning, - add to end.	"ASIS" (currrent position).
READONLY		Write protection (if specified, the file cannot be deleted when closing).	Unprotected when writing
RECL= RECORDSIZE=	*number*	Record length in the logical unit in the case of direct	Depends on the value specified in

		access, or maximum length in the case of sequential access (*number* is a positive integer).	the keywords: STATUS, ORGANIZATION and RECORDTYPE.
RECORDTYPE=	"FIXED" "VARIABLE" "SEGMENTED" "STREAM" "STREAM_LF" "STREAM_CR"	Recording structure: - fixed (all records will be of identical length), - variable, - segmented, - stream, - stream with line feed, - stream with carriage return.	Depends on keyword values: ACCESS, FORM.
SHARE=	"COMPAT" "DENYNONE" "DENYWR" "DENYRD" "DENYRW"	Controls how other processes can access the logical unit simultaneously on a network: - compatible, - without restrictions, - no writing, - no reading, - no read and write.	"DENYNONE" (no restrictions).
SHARED		Shared access to the logical unit (to the file).	Not shared.
STATUS= TYPE=	"OLD" "NEW" "REPLACE" "SCRATCH" "UNKNOWN"	State of the logical unit (of the file) when opened: - existing (if it does not exist, an error is obtained), - new (if already exists, generate error), - overwriting a file, - temporary (deleted after closing), - unknown (opened if exists, created if does not).	"UNKNOWN" (opened if exists, created if does not).
UNIT=	*number*	The logical unit number (associated with the desired file or device) being accessed (*number* is a positive integer).	Not implicit (the logical unit number can be specified without the UNIT= keyword if it is the first parameter in the parentheses).
USEROPEN=	*name*	Option for a user program.	No option.

Disconnection of the logical unit (in the case of files it means closing them) can be specified by the CLOSE executable instruction, the syntax of whichis as follows:

CLOSE (*parameter*[, *parameter*]…)

where *parameter* is of the form *keyword*=*value* (each *parameter* can be specified only once within the list in parentheses).

The CLOSE statement will also cause the <EOF> (end-of-file) to be recorded (written) when the unit is disconnected (file is closed).

Table of parameters in the CLOSE statement (in alphabetical order, the blue ones are not accepted by the G95 compiler):

keyword	*value*	Explanation	Implicit value
DISPOSE= DISP= STATUS=	"SAVE" "KEEP" "PRINT" "DELETE" "PRINT/DELETE" "SUBMIT" "SUBMIT/DELETE"	The state of the logical drive (usually the file) on close: - save, - keep, - print, - delete, - print and then delete, - invokes a process to execute the file, - invokes a process to execute and then delete the file.	"SAVE" (saving the contents of the logical unit).
ERR=	*label*	The label of the instruction to jump to in case of an error when disconnecting the logical unit (*label* is a positive integer).	Not implicit, no default jump.
IOMSG=	*variable*	Returns the contents of the *variable* (which is a CHARACTER scalar) in a message.	Not implicit.
IOSTAT=	*variable*	Returns a scalar INTEGER value in the variable, which indicates the success (or failure) of closing the logical unit. If the logical unit was successfully closed, the value of the variable is 0.	Not implicit.
UNIT=	*number*	The logical unit number (associated with the desired file or device) to disconnect (*number* is a positive integer).	Not implicit (the logical unit number can be specified without the UNIT= keyword if it is the first parameter in the parentheses).

Caution: A file opened with the "SCRATCH" specification cannot be saved, printed (displayed) or sent to a process ("SUBMIT"), such an attempt will generate an error at runtime, and if "READONLY" was specified when opening, the file cannot be deleted on disconnection (close). A read-only file does not necessarily need to be closed, but a file whose contents have been changed (written to) must be closed using the CLOSE statement, otherwise it may be stuck with inaccessible contents when the program finishes. Writing through a buffer, if it has not been explicitly emptied (by the effect of the CLOSE statement), then it is not certain that all records have been transferred, and at the end of the program run there will be no one to manage the contents of the buffer (resulting in the computer's memory being filled with unnecessary data).

Example:	Explanations:
OPEN(3,FILE="TEST.DAT",STATUS="OLD") READ(3,*)n,m CLOSE(3) DIMENSION A(10,10) CHARACTER(12) name PRINT *,"data file name: "	Open the existing TEST.DAT file associated with logical unit number 3, then read the values of variables N and M from this file and disconnect the logical unit (close the file). Declare an array of 10x10=100 positions and the NAME entity with 12 positions (characters). Note CHARACTER, the letter C in the first column

	marks comment!
`3  READ(*,"(A)")name` `OPEN(1,FILE=name,STATUS="OLD",ERR=9)`	Read the file name into the NAME variable. Open the (existing) file associated with logical unit 1, from which the data will be read (see the comments in the adjacent column marked with an exclamation mark). If the file does not exist, it jumps to the instruction labelled 9.

`! get the number of rows for A`
`  READ(1,*) nl`
`! get the number of columns for A`
`  READ(1,*) nc`
`! read the elements, one row at a time`
`  DO i=1,nl`
`  READ(1,*)(A(i,j),j=1,nc)`
`  ENDDO`
	Use an implicit loop (J=1,NC) inside an explicit loop (I=1,NL) to read elements from an array.

`  …` `OPEN(2,FILE="R.DAT",STATUS="UNKNOWN")`	Opening the R.DAT file associated with logical unit 2 (if the file does not exist, it is created, and if it exists, it is opened and its contents are overwritten).

`! Write the title to the R.DAT file`
`  WRITE(2,*)"Array A:"`
`! Write the elements, line by line:`
`  DO i=1,nl`
`  WRITE(2,*)(A(i,j)," ",j=1,nc)`
`  ENDDO`
	Write the elements of the array A, line by line, with a blank (" ") after each element.

`CLOSE(2)` `  …`	Close the R.DAT file (disconnect logical unit number 2). Logical unit number 1 has not been modified and will be automatically disconnected when the program ends.

`9  PRINT *,"file not found!"`
`GOTO 3`
`  …`
	If the data file is not found, after displaying the specified message, an attempt will be made to read its name again (jumping to the instruction labelled 3).

There are other additional instructions for handling files (coloured ones not recognised by the G95 compiler), such as:

Instruction syntax:	Explanations:
`UNLOCK([UNIT=]`*u*`[,ERR=`*label*`])` or `UNLOCK` *u*	Unlock a file (after associating it with a logical unit through the `OPEN` statement), where *u* is the number of the logical unit.
`REWIND([UNIT=]`*u*`[,ERR=`*label*`])` or `REWIND` *u*	Repositioning to the beginning of the file associated (previously by the `OPEN` statement) with logical unit number *u*.
`REWRITE([UNIT=]`*u*`[,[FMT=]`*f*`][,ERR=`*label*`])` *list*	Rewrite a record to the current position in the file associated with logical unit number *u* (via a previous `OPEN` statement), with format specification *f* (if specified).
`ENDFILE([UNIT=]`*u*`[,ERR=`*label*`])` or `ENDFILE` *u*	Write the end of file marker to the file associated with logical unit number *u* (accessed via a previous `OPEN` statement).

Format descriptors

Format descriptors are like templates applied to input or output data. They are usually used through the format specification, which has the following syntax:

label `FORMAT(`*descriptor_list*`)`

however, descriptors can also appear in quoted form within read or write statements.

There are two categories of descriptors: for data editing and for controlling formatting. They will be presented below in separate tables, with examples, using the following notations:

n – number of pieces;
w – descriptor length (total number of positions in the respective field);
m – minimum number of positions requested (of the total number), has effect on output only;
d – number of positions for the decimal part (of the total number);
e – number of positions for the exponent (of the total number);
c – character, respectively [*c*...] other optional characters;
□ – space (blank character) in examples.

Table of descriptors used for data editing (in alphabetical order):

Syntax:	Destination:	Examples and comments:
[*n*]A[*w*]	Alphanumeric data (CHARACTER)	Input: / Format: / Entity type: / Value: ABC_D / A5 / CHARACTER(1): D ABC_D / A5 / CHARACTER(3): C_D ABC_D / A5 / CHARACTER(6): ABC_D□ Value: / Format: / Output (5 positions): ABC / A5 / □□ABC ABCDE / A5 / ABCDE ABCDEFG / A5 / ABCDE
[*n*]B*w*[.*m*]	Binary numeric data	Input: / Format: / Value (in decimal form): 1001 / B4 / 9 (all 4 positions read) 1001 / B2 / 2 (only the first 2 positions read) 1001 / 2B2 / 2 și 1 (2 distinct values) Value: / Format: / Output: 13 / B5 / □1101 0 / B2 / □0 0 / B2.2 / 00 If *w*=0, as many positions as required to display the value will be used at the output (*w*=0 is not allowed at the input).
[*n*]D*w*.*d*	Numerical data in double precision: REAL(8) i.e. DOUBLE PRECISION, or COMPLEX(8), i.e. DOUBLE COMPLEX	Input: / Format: / Value (double precision): 123.456E3 / D9.3 / 123456.0D+0 12345678 / D6.2 / 1234.56D+0 123.45678 / D7.3 / 123.456D+0 As can be observed, *w* positions are read from the input, of which *d* positions for the decimal part (from the decimal separator to the right – if there is no decimal separator at the input, then the decimal part will result considering *d* positions at the end of the *w* read). The "D+0" mark at the end only indicates that the values will be obtained in double precision. Value: / Format: / Output: 123456.789 / D11.2 / □□□0.12D+06 0.0363 / D10.3 / □0.363D-01 -0.5555 / D10.3 / -0.556D+00 The display will result in *w* positions, of which *d* positions for the decimal part, but it should be noticed that 1 position will be

<table>
<tr><td></td><td></td><td>consumed for the sign of the value, 1 more for the decimal separator (dot), 1 position for the letter of the descriptor (D), the last 3 positions for the sign and value of the exponent.
If we consider that the first significant digit will be the first decimal place, it follows that it is advisable that $w-d > 6$. If this condition is not met, format overflow will occur (asterisks will be displayed on the w positions).</td></tr>
<tr><td>[n]Ew.d[Ee]</td><td>Numeric data in exponential format (REAL or COMPLEX)</td><td>Input: Format: Value:
□□123.45□□ E10.2 123.45
123456789 E9.3 123456.789
123.456D3 E9.3 123456.0 (simple precision!)
As with the previous descriptor, w positions are read from the input, of which d positions for the decimal part (from the decimal separator to the right – if there is no decimal separator at the input, the decimal part will result considering d positions at the end of the w read). In case of reading double precision values with this descriptor (or with other usable descriptors except D), a value converted to single precision will be obtained.

Value: Format: Output:
123456.789 E11.5 0.12345E+06
-0.5555 E12.3E3 □-0.556E+000
0.0363 E5.2 ***** (format overflow!)
The display will result in w positions, of which d positions for the decimal part, but it should be noticed that 1 position will be consumed for the sign of the value, 1 more for the decimal separator (dot), 1 position for the letter of the descriptor (E), the last 3 positions for the sign and value of the exponent.
If we consider that the first significant digit will be the first decimal, it turns out that $w-d > 6$ [+(e−2)] (where e is the number of digits of the exponent). If this condition is not met, format overflow will occur (asterisks will be displayed on the w positions).</td></tr>
<tr><td>[n]ENw.d[Ee]</td><td>Numeric data in exponential ”engineering” format (REAL or COMPLEX)</td><td>Input: Format: Value:
123.45E+03 EN10.2 12345.0
-12345678 EN9.3 -12345.678
123.456D3 EN9.3 123456.0 (simple precision!)

Value: Format: Output:
123456.789 EN11.2 □123.46E+03
-0.5555 EN7.1 ******* (format overflow!)
0.0363 EN12.3 □363.000E-04
When displayed, the decimal point will be after the first 3 digits.</td></tr>
<tr><td>[n]ESw.d[Ee]</td><td>Numeric data in exponential ”scientific” format (REAL or COMPLEX)</td><td>Input: Format: Value:
□□□1.234E+03 ES12.3 1234.0
-10.234E-03 ES11.3 -0.010234

Value: Format: Output:
123456.789 ES11.2 □□□1.23E+05
-0.5555 ES10.3 -5.555E-01
0.0363 ES12.3 □□□3.630E-02
On display the decimal point will be after the first significant digit.</td></tr>
<tr><td>[n]Fw.d</td><td>Numeric data (REAL, F stands for ”Float”)</td><td>Input: Format: Value:
12345678 F8.5 123.45678
-12345678 F8.2 -1234.56
24.77E+2 F8.2 2477.0</td></tr>
</table>

		Value:	Format:	Output:
		2.3547188	F8.5	□2.35472
		325.03	F5.2	***** (format overflow!)
		-0.2	F5.2	-0.20
[*n*]G*w*.*d*[E*e*]	Intrinsic type data (G stands for "Generic")	Input:	Format:	Value:
		-0.05566	G10.3	-0.05566
		123456	G10.3	123.456
		123456.789	G10.3	123456.79
		Value:	Format:	Output:
		-45.66	G11.3	□-4.566E+01
		123456	G10.3	□□□□123456
		123456.78	G10.3	□0.123E+06
		Remarks: Descriptor G can be used for any of the intrinsic type values. If 0 is specified for *w*, the actual value of *w* will be chosen by the processor (in such cases only G0 or G0.*d* may be specified). If *w* is different from 0, then the value for *d* must also be specified. In the case of INTEGER, CHARACTER and LOGICAL values the value specified by *d* will be ignored, the descriptor will behave as the one corresponding to these values (I, A and L).		
*w*H*c*[*c*...]	Hollerith constants (CHARACTER)	Input: It is not recommended to use, because the content of the constant can be modified (the G95 compiler will give an error message!)		
		Format:		Output:
		10H#'-'abc"3		#'-'abc"3
		21H "Hollerith" constant		□"Hollerith"□constant
		Remark: Although these constants were originally defined to contain up to 2000 characters, the number of characters can be between 1 and 32767 ($2^{15}-1$) on 32-bit platforms, or between 1 and 2147483647 ($2^{31}-1$) on 64-bit platforms.		
[*n*]I*w*[.*m*]	Integer numeric data (INTEGER)	Input:	Format:	Value:
		-1234	I4	-123
		□□□123	I6	123
		1234.6	I6	Error! (not INTEGER)
		Value:	Format:	Output:
		0	I3	□□0
		0	I3.0	□□□
		1	I3.2	□01
		-123	I3	*** (format overflow!)
		1.2	I4	Error! (not INTEGER)
[*n*]L*w*	Logical data	Input – logical values written in the following forms are accepted, including lowercase (not just uppercase): .TRUE. or .T or T or if the first characters in the input are .T or T (for true), or .FALSE. or .F or F or if the first characters in the input are .F or F or the content is from space/ blanks (for false).		
		Value:	Format:	Output:
		.TRUE.	L7	□□□□□□T
		.FALSE.	L1	F
		□□□	L3	□□F
		Only 1 character (T or F) will be output regardless of the length *w* specified.		
[*n*]O*w*[.*m*]	Integer octal numeric data (with base in 8)	Input:	Format:	Value (decimal):
		1111	O2	9

		1111	O4	585
		□11□	O4	9
		191	O3	Error! (9 is not octal)
		12	O0	Error! (*w* must be positive)
		Value (decimal):	Format:	Output:
		11	O6.4	□□0013
		-11	O6	****** (format overflow!)
		-11	O12	□37777777765
		1.5	O11	□7760000000
		81	O0	121
		If *w*=0, as many positions as required to display the value will be used at the output (*w*=0 is not allowed at the input).		
[*n*]Z*w*[.*m*]	Integer hexadecimal numeric data (with base in 16)	Input:	Format:	Value (decimal):
		A2F	Z3	2607
		−A2F□	Z5	-2607
		3.A2F	Z5	Error! (invalid decimal point)
		Value (decimal):	Format:	Output:
		3033	Z5	□□BD9
		16	Z5.4	□0010
		-10	Z8	FFFFFFF6
		1.1	Z8	3F8CCCCD
		2.5	Z0	40200000
		If *w*=0, as many positions as required to display the value will be used at the output (*w*=0 is not allowed at the input).		
’*c*[*c*...]’ or ”*c*[*c*...]”	Quoted alphanumeric constants (CHARACTER)	Input: not applicable (cannot be used for entries).		
		Format:		Output:
		’aBc’’DD:’	(apostrophe doubled inside)	aBc’DD:
		’aBc”DD:’	(contained quotation mark)	aBc”DD:
		”abcD’#”		abcD’#
		”ab’cD’#”	(quote within quote!)	Error!

Table of control descriptors:

Syntax:	Meaning:	Examples and comments:
BN BZ	BLANK NULL BLANK ZERO	BN will have the effect of ignoring the spaces in the number fields. BZ will have the effect of ”replacing” the spaces in the numeric fields with 0 digits. Input: Format: Value: □1□□ BN,I4 1 □1□□ BZ,I4 100 1□23 BZ,I4 1023
*k*P	Power *k* is a scaling factor, with value in the range [-128, +127]	Allows the interpretation of numeric values with decimals, using the scale factor *k* for descriptors D, E, F and G when these values do not explicitly contain an exponent. On inputs, a positive *k* value will have the effect of moving the decimal separator to the left, and a negative value to the right (on outputs, the effect will be the reverse). The descriptor P need not necessarily be separated by a comma from the descriptor to which it refers, but must precede it. For example, the following specifications will have the same effect, the scale factor being associated with the first real number descriptor following it in the list (E10.3): (2P,I4,E10.3,F8.2) (I4,2P,E10.3,F8.2)

		(I4,2PE10.3,F8.2) Input: Format: Value: □□□37.614□ 3PE10.5 0.037614 □□□37.614□ -3PE10.5 37614.0 123.45 2PF8.3 1.2345 123.45 -2P,F8.3 12345.0 Value: Format: Output: -170.139 1P,E10.3 -1.701E+02 -170.139 -1PE10.3 □-0.02E+04
Q	Quantity	Returns the number of characters (positions) left unprocessed from the input (although this is usually supported for compatibility reasons, it is not part of newer versions of Fortran and the G95 compiler will report it as an error). Source code: READ(*,'(Q)')nr PRINT 2,nr READ(*,'(Q)')nr PRINT 2,nr 2 FORMAT('Buc: ',I2) END Input: abcdefg Display: Buc: 7 Input: 55aa Display: Buc: 4
S SP SS	Sign Sign Positive Sign Suppress	SP will cause the + sign to be displayed in front of positive values and SS will inhibit it. S acts as a switch between SP and SS.
T*n* TLn TR*n*	Tab Tab Left Tab Right *n* – tab position	With descriptor T, the position *n* in a line is indicated, from which reading (or to which writing) is desired. Assuming that the following string will be entered from the keyboard: 123456789ABC to be read with the sequence: CHARACTER(3) C1,C2 READ(*,5) NR,C1,C2 5 FORMAT(T7,I3,T1,A3,T10,A3) the values will result: NR=789; C1="123" și C2="ABC". TR*n* allows specifying the n^{th} position to the right from the current position and TL*n* to the left (*n* being a positive number). When using TL, if *n* is greater than or equal to the current position, then positioning will be done on the first character in the row.
[*n*]X	Determine the jump over *n* positions in the current line	On input it will cause *n* positions to be ignored, and on output it will have the effect of printing *n* spaces (if it appears at the end of the descriptor list, then it has no effect. In the example the effect is highlighted by marking □ on display): Source code: PRINT 4 READ 3,nr PRINT 4,nr 3 FORMAT(2X,I2) 4 FORMAT("number:",1X,I2) Display: number: Input: 1234 Display: number:□34
$ \	Suppress the jump to a new line (suppress <*LF*>).	It will cause the cursor to remain at the last current position (<*LF*> is short for *Line_Feed*). The $ variant is newer, but not part of the standard, and the \ variant is the "classic" one (the G95 compiler supports both). Whichever variant is used, the descriptor must be the last in the list to which it belongs.

		Source code:
		`PRINT 5,"nr:"`
		`READ *,nr`
		`PRINT 4,nr`
		`5 FORMAT(A,$)`
		`4 FORMAT("number:",1X,I2)`
		Display+Input (12):
		`nr:12`
		Display:
		`number:□12`

[*n*]/	Induces *n* new line jumps (induces *n* pieces of <*LF*>)	It can also be used without n, e.g. (3/) is equivalent to (///), without the need for separating commas. In the following example it will insert a new line feed before displaying "number", then insert 2 more new line feeds:
		Source code:
		`PRINT 5,"nr:"`
		`READ *,nr`
		`PRINT 4,nr`
		`5 FORMAT(A,$)`
		`4 FORMAT(/"number:",2/,3X,I2)`
		Display+Input (12):
		`nr:12`
		Display:
		`□`
		`number:`
		`□`
		`□□□12`

:	Ends descriptor control in the absence of input/output list items	In the following example, in the absence of items to display, the descriptor will cause the "j2" part to be ignored:
		Source code:
		`PRINT 1,3`
		`PRINT 2,14`
		`1 FORMAT("i",I2,1X,"i2",I2)`
		`2 FORMAT("j",I2,:,1X,"j2",I2)`
		Display:
		`i□3i2□□`
		`j14`

The format specification may also be composed of string (character) expressions. The following example shows how it might apply for N pairs of descriptors of the form (`I2,1X`), assuming 1< N <9:

Example:	Explanations:
`CHARACTER fm(10)`	Declare the FM string with 10 positions (to be used as format specification).
`INTEGER j(9)`	The variable J will have 9 positions (will be a vector) and will contain the values to be displayed with descriptors of type `I2`.
`PRINT *,"nr. (1-9): "` `READ(*,*)n`	The quoted string is displayed and on reading the number entered (of desired pieces) is stored in variable N.
`k=48+n`	The character table position number of the digit corresponding to the number of pieces (in variable N) is composed, obtaining the digit character representing that value.
`fm="("//ACHAR(k)//"(i2,1x))"`	By concatenation and using the intrinsic function `ACHAR` (which returns the character at position K in the character table) an alphanumeric string is composed and assigned to the variable FM, which will be the format descriptor with N pairs of fields of type `I2` (integer two positions) and `1X` (one space) for the N values.
`PRINT *,"the ",n," values: "` `READ *,(j(i),i=1,n)`	The quoted string (including the value of N) is displayed, then the values corresponding to the N positions of the vector J are read (by implicit loop).
`WRITE(*,fm)(j(i),i=1,n)`	The N positions of the vector J are displayed (also by implicit loop) using the format specification stored in the FM variable as an alphanumeric string.
`END`	