A Tutorial on the

EMME/2 Macro Language

by

Heinz Spiess, EMME/2 Support Center

September 1996

Abstract

One of the most powerful features of the EMME/2 transportation planning system is its macro language. Macros are not only an ideal tool for the automation of complex or repetitive tasks, but, if used judiciously, may also be an invaluable help when it comes to the standardization, documentation and transferability of a once developed procedure.

While, even for a novice EMME/2 user, it is quite easy to write and use simple "save and recall" type macros, mastering the macro language down to its most intricate details is an art that distinguishes the expert user.

This tutorial is aimed at those EMME/2 users that have already some experience with macro writing, but would like to improve their skills and learn more about some of the advanced possibilities the macro language offers. The text is not intended to replace the standard documentation, rather it aims to complement it, by providing more detailed explanations and explicit examples on some selected topics.

This text was initially written as course notes for the two-day course on advanced EMME/2 macro writing, organized by INRO Consultants Inc. in Montreal, December 8 & 9, 1993. The current version contains adaptations to cover new features of the macro language which are included in EMME/2 Release 8.

Copyright ©1993 – 1996. Heinz Spiess, Aegerten, Switzerland. All rights reserved.

Contents

Copyright:

After having used this tutorial at several occasions as course notes for a 2-day course on "Advanced EMME/2 Macro Programming", I have now decided to make it available to all EMME/2 users by posting it on my Web site (`http://www.spiess.ch/emme2`). This way, all EMME/2 users that are interested in improving their macro writing skills can use this material for self-study and can test their knowledge of the EMME/2 macro language by solving the exercises at the end of each section.

Note that this material is only made available for the purpose of the **individual use** by interested EMME/2 users. Thus, unless you have obtained my explicit prior permission, you are not allowed to redistribute this tutorial (but it's okay to pass a copy to a friend), nor to use it or parts of it when teaching EMME/2 courses to other EMME/2 users.

This tutorial is also accessible in HTML format at `http://www.spiess.ch/emme2`.

1 Introduction

Macro writing is not a theoretical science, rather it is a "rough" trade which you can only learn by getting yours hands "dirty". For this reason, the most important part this tutorial are the exercises which you will find at the end of the sections 2 – 8. These exercises were chosen in such a way that they can be solved in a short time (not more than 15 minutes for most of them), provided that you have understood well the concepts explained in the corresponding section. Thus, in your very own interest, really try to solve the exercises yourself. Even if you do already have access to the solutions (which you normally don't, see section 11), it is worth while to give it a good first try on your own.

The following sections are structured in a more or less logical way, starting with a general overview and then proceeding from the simpler to more complex aspects of macro programming. The only section with does not really fit into this order is section 9 which explains how to debug macros. As solving the exercises has a lot to do with macro debugging (at least for me!), it might be worth while to check into section 9 after having having lived through the pains of debugging the first couple of macros without using any special debugging commands.

In the next section, we will give a brief overall summary of the macro language. In each of the remaining sections, a particular aspect of advanced macro programming is looked at in more detail.

2 Macro Command Language Summary

This section is not intended as a general introduction to EMME/2 macros. Rather, it was written with the intention of being "a quick refresher" for users that have already worked with the macro language, but do not have the same degree of familiarity with all features of the language.

In order to get an idea what the concept of 'macros' in the context of EMME/2 is all about, let us first look at the general framework how the operation of the EMME/2 package is controlled by the user.

The basic method of telling EMME/2 what to do is based on the notion of *dialog*. The dialog consists of a sequence of interactions in which the system asks *questions*, which in turn are *answered* by the user. The structure of this dialog is quite simple, as there are only three well defined types of questions that make up the dialog:

YES/NO QUESTION: This question is identified by the question mark "?". Possible answers are limited to "yes", "no", "restart" (back to beginning of current module) or "quit" (back to main menu).

SELECT QUESTION: A choice of two or more alternatives is presented to the user, each one preceded by a sequential number. The user answers by choosing one of the alternatives by entering the corresponding number, or she has the possibility to "restart" or "quit".

ENTER QUESTION: This type of question is always used when the user is prompted to enter one or several textual or numerical data items. The question terminates with an equal sign "=" as prompt for the user to enter the data. If the "=" appears on the same line with the question text, the answer is limited to a single line, but if the "=" appears on a separate line underneath the question text, the answer is composed of multiple input lines, terminated by an empty line.

When EMME/2 is used interactively, all answers to the questions asked in the dialog are provided by the user, who is typing them at the keyboard as he goes along. The macro language provided in EMME/2 is simply a mechanism which generates these answers automatically, according to a macro script. In other words, whatever a user can do by operating the EMME/2 dialog interactively can also be implemented in a macro.

Therefore, in its simplest form, a macro consists of just a straight sequence of pre-defined answers that are passed back, one after the other, to the EMME/2 dialog. However, in most real-life applications, this would not be good enough, as the answers that need to be generated by the macro are seldomly exactly the same from one run to the other. To address this need, the macro language incorporates many features that allow the implementation of "intelligent" macros, such as:

- Macro *parameters* can be used to pass arguments to the invoked macro.
- A general *substitution* mechanism can be used to define context dependent parts of the answers.

- The *flow control* in the macro can be defined by the use of labels, branches and conditionals.
- A set of *integer, floating point and text registers* can be operated upon in order to implement general purpose computations. ← *Rel.8!*
- External programs and procedures can be bound into a macro by the directly issuing *external commands* to the underlying operating system.
- Error detection and recovery mechanisms allow trapping of *error conditions*.
- Macros can invoke other macros as sub-procedures, allowing the implementation of a *hierarchical calling structure* of arbitrary depth.
- The standard dialog output of EMME/2 can be suppressed during the execution of a macro and the macro can generate its own messages and interact directly with the user. This allows a macro to implement its own *front end*.
- Macros can directly access the contents of *scalars* (read-only) and the elements of the `get()/put()` *stack of the expression evaluator* (read/write). ← *Rel.8!*
- If needed, the macro language provides access to *system specific* and *internal* data bank parameters.
- A special *single step macro tracing mode* helps the macro writer to write and debug macros more efficiently, or trouble-shoot them in case of problems later on.

In the remaining part of this section, a brief overall summary of the EMME/2 macro language is given, based on the implementation in EMME/2 Release 8.

Various types of macro registers are available. Some of them are read-only registers which contain system defined parameters, others are writable by the macro, so that they take the role of programming variables, which can be used for computations, counters, etc. All numeric registers are restricted to integer values. Text registers have a maximum length of 128, but since the maximum macro line length is also limited to 128, it is normally not possible to attain this maximum. The following table gives an overview of all available registers.

Macro Registers

register:	type:	access:	contents:
b	integer	read-only	Batch mode indicator (1:interactive, 2:batch). In batch mode all reports are written to the report file, all plots are written to the plot file, without corresponding select questions and regardless of the settings of switches 0 and 1. Also, in batch mode, no change of range is possible after a plot is generated with modules 2.43, 3.16 or 4.13, the module returns to the primary select right after the plot is terminated.

Macro Registers *(... continued)*			
register:	type:	access:	contents:
d	integer	read-only	Date register. Contains a six digit integer value of the form YYMMDD, where YY indicates the year (0-99), MM the month (1-12) and DD the day of the month (1-31).
e	integer	read-only	Number of last error. This register contains the number of the last error (warning or fatal) which has occurred, as described in Appendix A of the User's Manual. It can be used to take the corrective action after an abortive error condition has been detected, or also to determine, if errors occurred while reading a file using batch input.
f	integer	read-only	Scenario status flags for current scenario. This register contains a bit pattern defined as follows: bit 0: protected against forced module execution bit 1: protected against network modifications bit 2: protected against scenario deletion bit 5: ready for an auto assignment bit 6: ready for a transit assignment bit 10: contains valid auto assignment bit 11: contains valid transit assignment
i	integer	read-only	Bit pattern containing the current state of switches 0-31. See User's Manual for definition of the individual switches.
m	integer	read-only	Current module number. This register contains the number of the currently active module as a 3-digit integer value.
q	integer	read-only	Type of current dialog question. This register can be used to determine the type of question the macro is currently expected to answer. A value of 1 indicates a Yes/No question, a value $N > 1$ indicates a Select question with N alternatives, and a value of 0 is used for Enter questions. A value of -1 indicates an extra input line generated when bit 6 of the control register o is set (see section 6 for more information). The register q is very useful to synchronize a macro in situations where the dialog varies depending on the current context (e.g. the dialog of an assignment preparation is slightly different if the scenario already contains a valid assignment).
s	integer	read-only	Number of the current scenario.

Macro Registers *(... continued)*			
register:	type:	access:	contents:
v	integer	read-only	Version of current module. This register contains an integer value which indicates the major release level (hundreds and up) and the minor update level (last two digits) of the current module.
x,y,z	integer	read/write	General purpose integer registers. These three registers can be used by the macro to hold or compute any type of numeric information.
p	integer	write addr. read value	The `p` register provides a convenient mechanism to access system, global and scenario parameters. When a value is written into the `p` registers, it is interpreted as the address of one of these parameters. When, on the other hand, the value of the `p` register is read (e.g. in the substitution %p% or in the conditional `~?p>0`), the value of the corresponding parameter is used. See section 7 for further discussion.
o	integer	read/write	The `o` register contains a bit pattern, which can be used to control the various aspects of the dialog output. See section 6 for further discussion.
r1-r250	real	read/write	General purpose single precision floating point registers. These registers can be used to hold and manipulate any kind of numeric information. Instead of specifying the register number explicitly, it is also possible to use one of the registers `x`, `y` or `z` as index register, e.g. if `y=25`, accessing `ry` is the same as accessing `r25`.
g1-g250	real	read/write	These single precision floating point registers are used to directly access the elements of the `get()/put()` stack. This allows direct two-way interactions between macros and expression evaluations. Instead of specifying the register number explicitly, it is also possible to use one of the registers `x`, `y` or `z` as index register. Since there is only one `get()/put()`stack, the gN registers are global, i.e. shared by all levels of macro invocation. Note that these registers are reset to zero each time a new module is started.

Macro Registers *(... continued)*			
register:	type:	access:	contents:
t0-t9	text	read/write	Ten text registers of up to 128 characters are available. The register `t0` is special in that it always corresponds to the current macro calling parameters %1% %2% %3% ... . Thus, changing the contents of `t0` will always affect the macro parameters, and, vice versa, modifying the macro parameters (e.g. with the ~% command) will affect `t0`. The register `t9` is a global register, i.e. its value is common to all levels of macro invocation, so that this register can be used to pass return values from the called macro to the calling macro.

As explained above, the basic task of a macro is to provide "automatic" answers to the EMME/2 dialog. Since often these answers depend on the context, an important component of the macro language is a general substitution mechanism which can be used to implement such dependencies. Those parts of the answers that are not fixed, but have to be replaced by context dependent information at the time the macro is executed, are coded into the macro text as *substitution keys*. These keys are always surrounded by % signs (e.g. %1% or %ms77.6%), in order to make them easily recognizable and to prevent ambiguities with the fixed parts of the macro text. At execution time, all valid substitution keys are replaced by their corresponding current value.

The following table contains a list of all possible substitutions:

Macro Substitutions	
key:	substituted by:
%0%	Current number of macro parameters. This number is always between 0 and 9. Since at most 9 parameters are available for direct substitution, the number does not exceed 9, even if the number of actual parameters is larger than 9. Often %0% is used to determine if a macro has been called with the correct number of parameters.
%1% ... %9%	Macro parameters 1 to 9. If a parameter is not defined it is substituted by an empty string. Note that the text register `t0` always corresponds to the current macro parameters.
%*reg*%	Value of register *reg*.
%*reg*_*W*%	Value of integer or floating pointregister *reg* using a predefined field width of *W* characters. If the value does not completely use the specified field width, it is adjusted to the right.
%*freg*.*D*%	Value of integer or floating point register *freg* using *D* digits after the decimal point.

Macro Substitutions *(... continued)*	
key:	substituted by:
%*freg.D_W*%	Value of floating point register *freg* using *D* digits after the decimal point and a predefined field width of *W* characters.
%u%	User initials, as specified at session start.
%t*N.L.*%	The first *L* characters of text register t*N*.
%t*N.-L.*%	The contents of the text register t*N* without the first *L* characters.
%ms*N*% %ms*N_W*% %ms*N.D*% %ms*N.D_W*%	The substitution key is replaced by the value of a scalar. *N* can be a number between 1 and 99, or alternatively, one of the letters x, y or z. In the latter case, the scalar is addressed indirectly by using the scalar number which corresponds to the current value of the corresponding register. By default, automatic formatting is used, which eliminates redundant trailing zero decimals. If needed, the scalar identifier can optionally be followed by a decimal point and a digit between 0 and 9, and/or an underscore character followed by a field width. In this case, the format conversion is forced to use the corresponding number of digits after the decimal point and/or the specified field width. Note that when *D*=9, the default automatic format conversion is used.
%ms*N*.n%	Same as above, but the name of the scalar is substituted instead of its value. Any trailing blanks are removed from the 6 character name.
%ms*N*.d%	Same as above, but the description of the scalar is substituted instead of the scalar value or name. Trailing blanks are removed from the 40 character description.
%%%	This sequence is substituted by a single %. This is often useful for preventing the substitution of a valid substitution key.

Before being executed, each line of the macro file is checked for any of the above substitution keys, and the necessary substitutions are performed. This holds for macro lines that contain dialog answers, as well as lines containing macro commands. The only exceptions are the macro address labels, where, for the sake of efficient branching, no substitution is applied.

All lines in the macro file that do not start with the tilde character (~) are assumed to be dialog answers. After the applicable substitutions have been made (if any), they are passed back to the calling module.

Lines in the macro file which start with a tilde character (~), or alternately with a blank

followed by a tilde character, are called *macro commands*. These commands are used to control the macro's operation without necessarily generating a dialog answer. The following list summarizes the available commands and their syntax.

Macro Commands	
command:	description:
~/*comment*	Comment line. When a comment is encountered during the processing of the macro, the comment line is copied to the screen and the processing continues without further actions. If the bit 1 of the o-register is set to OFF, the comment command is copied as a whole to the screen, if set to ON, the leading ~/ characters are omitted, so that the text will not be recognizable as a macro comment.
~~/*comment*	Comment with no line-feed. This variant of the comment command is similar to ~/, but the comment is output without finishing the line with a line-feed character. This allows a macro to write a line on the screen in several steps, which can be useful to implement progress reports as the macro executes.
~**prompt string*	Read line from keyboard. This command will interactively read one line from the keyboard. An optional text string can be given after the command, which will be displayed as a prompt on the same line on which the answer is to be typed in by the user. If no prompt string is specified, the string ">>" is used by default. Normally, the line which is entered interactively by the user is passed directly back to the calling module as answer to the current question. However, if the command appears to the right of a "set text register" command (e.g.: `~t2=~*Enter: Matrix to symmetrize=`), the command can be used to save the text entered by the user into the specified text register.
~:*label*	Define address label. This command is used to define a branch target address in the macro. The execution of a branch command with the same label string (~$*label*) will cause the macro to be continued at this address. The label strings of the branch command must *exactly* match the string in the corresponding label definition command, including leading and trailing blanks! Note that the usual macro substitution (%...%) does *not* apply to this command (for efficiency reasons).

Macro Commands	*(... continued)*
command:	description:
~$*label* ~$>*label*	Branch to specified address label. In the simple branch statement (~$), the macro file is rewound to the beginning and the macro file is scanned until a corresponding ~:*label* is found. In the case of the forward branching (~$>), the macro file is scanned starting from the current position (i.e. is not rewound) for the corresponding label. If the target address is known to be located after the branch command, using the latter form is much more efficient, especially when dealing with large macro files. (In compound macro commands ~+... (see below) an "empty" branch command ~$, i.e. without specifying any label, can be used to branch back to the beginning of the compound statement.)
~%	Discard current first parameter %1% and shift remaining macro parameters by one position to the left. The second parameter will be moved into position 1 (%1% ← %2%) , the third into position 2 (%2% ← %3%), and so on. This command is useful when a macro does the same operation on all specified parameters. In this case, the operation is implemented for the first parameter using %1%, then the parameters are shifted using the ~% command, and this sequence is repeated as long as there are still macro parameters remaining.
~*reg*=*value*	This class of commands is used to initialize a writable text or integer register *reg* to a given value *value*. *reg* can be any of of the integer registers `p`, `o`, `x`, `y` or `z`, the floating point registers `r`*N* and `g`*N*, or the text registers `t0` – `t9`. Since this command is subject to the usual substitutions before being executed, it can also be used to copy values from one register to another , e.g. the command ~x=%z% copies the contents of the register `z` to register `x`.

Macro Commands	*(... continued)*
command:	description:
~*reg*⊗*value*	This class of commands is used to perform an integer arithmetic operation on a writable numeric register *reg*, where *reg* can be any of of the integer registers p, o, x, y or z, or the floating point registers r*N* and g*N*, The operator ⊗ can be any of + (addition), - (subtraction), * (multiplication), / (division), % (remainder of division), and only for integer registers | (bitwise OR) or & (bitwise AND). The first operand is always the current register value, while the second operand is the specified integer value *value*. The result of the operation is stored back into register *reg*. Since the command is subject to the usual substitutions before being executed, standard register substitution can be used for performing operations based on the value of another register. E.g. the command ~z+%y% adds the contents of register y to register z ($z \leftarrow z + y$).
~?*reg*⊕*value* ~?!*reg*⊕*value*	Conditional. This command is used to test the current contents of a register *reg* against a given value *value*. The following comparison operators ⊕ are available: < (less than), = (equal), > (greater than). For text registers, these comparisons are done using the standard ASCII collating sequence. For integer value registers the bitwise AND operator & is also available to test for bit masks. In the first form of the command, the line following the conditional is carried out only if the specified condition is true (skipped otherwise). In the second form (~?!), the condition is complemented, i.e. the following line is executed if the condition is false condition, and skipped if it is true. Note that the latter form in fact allows the implementation of the comparisons $\leq$, $\neq$ and $\geq$.
~?e ~?!e	Test for error condition. This command is used to detect if an error condition has (or has not) occurred since the last macro command. The line following the ~?e is executed only if an error condition has occurred, whereas the line following the ~?!e is executed only if no error condition has been detected. These two commands can be used to catch and correct error conditions which, if not caught in this way, would cause the macro processing to be interrupted immediately.

Macro Commands	*(... continued)*
command:	description:
~<*macro p1 p2 ...*	Call macro as a sub-procedure of the current macro. The specified macro *macro* is executed with the given parameters. Upon termination of the lower level macro, the current level macro continues its execution normally. Each level of macro invocation sees its own private set of registers. The values of these private registers are initialized to the current values of the same registers in the calling macro, but they can be modified by the called macro at will, without danger of disturbing the macros at higher levels. Text register `t9` and the floating point registers g*N* are exceptions to this rule, in that all levels of macro invocation share the same (global) value. Thus, these registers (in particular `t9`) can be used to pass return values back to the calling macro.
~!*external cmd*	Command escape to the operating system. The *external command* is passed to and executed by the underlying operating system. This command allows any kind of external programs and procedures to be integrated into a macro. Since the operating system depends on the installation, imprudent use of this command will result in macros that will only run in a particular operating environment.
~+*X....X....X....*	Compound macro command. This command can be used to "pack" several dialog answer lines and macro commands into a single physical line of the macro. The character immediately following ~+ is used as separator between the partial commands (here *X* is used, but any other printable ASCII character can be used). The compound command is not only useful to group dialog answers into logical groups (making the macro file considerably shorter and easier to read), but, immediately after a conditional, it can also be used to make the conditionals act on an entire group of answers or commands. Note that, except for label definitions (~:) and other compound statements (~+), all macro commands can be used as subcommands. The standard substitutions are applied twice, once for the compound command as a whole, and once at the level of each subcommand. Thus, care has to be taken if the registers used in the substitution are also modified within the compound command. An empty branch command (~$, no *label*) within a compound macro command will cause a branch to the beginning of the compound command. This type of branching is particularly efficient, since it does not imply scanning of the macro file to find the corresponding label.

Since a macro file is a regular ASCII text file, it can be edited using any text editor that the user is familiar with. Usually, a macro is created by first building an initial skeleton version of the macro using the "save macro" command ~>*macrofile p1 p2 p3 ...* (this mode of operation is also referred to as *macro learn mode*. The basic functions of the macro are then carried out interactively while EMME/2 is transcribing all dialog input into the designated macro file. Note that during this process it is already possible to define and use macro parameters, as well as any other substitution keys. This is done by specifying the values of the macro parameters which are to be used while the macro is being saved on the command line. Then, while interactively creating the macro, whenever a part of the input is not fixed, but needs to be substituted when the macro is executed later on, the user enters the substitution keys instead of their actual current values. The substitution mechanism will now do the necessary replacement even during the interactive macro definition. In this way, a macro is created which already contains the proper substitution keys. An empty save command (~> – no file name!) is used to terminate the saving of a macro in learn mode.

While a macro is created using the "save macro" command, it is also possible to enter directly some of the more simple macro commands, such as comments (~/...), register assignments and operations, label definitions and read line (~*). Other commands, such as conditionals, branching and compound commands, are not accepted in the macro learn mode and will have to be added later by explicit editing of the macro file.

EXERCISE 2.1: Write a macro sequence to answer the dialog generated by the question "`Enter: Matrix( mf )=`", which is asked whenever a full matrix is to be specified in which results are written into. The macro should work equally well whether the matrix already exists or not. If the matrix exists, the values must *not* be initialized. Newly created matrices are to be initialized to zero. Assume that the matrix identifier is stored in %1%. Matrix name and description are given by the macro context.

EXERCISE 2.2: When changing to a new release level, it is possible that existing macros may need to be adapted. How could a macro detect that it is being used with a higher release level than the one it has been tested with so far? Write the necessary macro statements to test for this situation and, if the release level does not match, display an error message and exit.

3 Hierarchical Macro Calls

For implementing complex procedures, it is sometimes useful to encapsulate individual subtasks into separate macros, which can be developed and tested independently, and which might sometimes be used as stand-alone macros for performing just the corresponding subtask. The macro which performs the complete procedure can now perform these subtasks by using "call macro" commands of the form ~<*subtask p1 p2 p3 ...* .

When a "call macro" command is encountered, the processing of the current macro is temporarily suspended and the called macro is executed as a subroutine of the calling macro. Once the execution of the called macro is terminated, the calling macro resumes operation at the line following the "call macro" command. The calling of sub-macros is not restricted to a single level, but it can be applied to as many levels as are required, thus allowing the implementation of an arbitrary hierarchy of macro calls. Even recursive invocations of the same macro are allowed.

While the principle of hierarchical macro invocation is straight forward, it is important to look at what exactly happens to the register and parameter values when one macro calls another macro. Since each macro uses the registers differently, it is important to avoid conflicts, e.g. as would result if a called macro would modify register values that the calling macro still needs for its continuation. Also a calling macro needs to pass argument values to the called macro, and, upon termination, the latter may need to pass back return values. In the remaining part of this section, we will look at these questions in detail.

In the implementation of hierarchical macro calls, registers are implemented in such a way that each macro is provided with its own, private set of registers. This implies that a macro does not need to worry about the register usage of the macros that it calls. Even if an invoked macro uses the same registers for storing its own information, the calling macro will still find its own register values unchanged upon the return of the called macro. This way, conflicting register usage between different macros is impossible.

When a macro is called by another macro, its (private) register values are initialized to the current register values of the calling macro. This allows that parameters can be passed from the calling to the called macro via the registers (in addition to the parameters that are specified on the call macro command itself). On the other hand, this also implies that a macro should never assume that the registers are initialized to zero (or blank for text registers), but always initialize the used registers explicitly.

When macro needs to pass a return value back to the calling macro, it can do so by storing the return value into the text register `t9`. As an exception to the rule explained above, the `t9` register is implemented as a global register. Therefore, all macros share the same value of `t9`, and a change to this register made in a lower level macro, is visible to the macros at the upper levels. If more than one return value are to be passed back, these can be assembled into the text register (which can hold up to 124 characters) and be disassembled back into its parts in the calling macro.

EXERCISE 3.1: Assume that you are writing a macro which needs to invoke existing macros which you do not know in detail and you suspect that they might not explicitly initialize the registers before using them, but (wrongly!) rely on them being already initialized to zero (resp. blank). Write a general purpose "wrapper" macro which can be used to invoke such macros.

EXERCISE 3.2: Write a recursive macro `factor` to compute the factorial $n!$, where n is given as macro parameter and the result is stored in `t9`.

4 Compound Macro Statements

With compound macro statements, several macro answers and/or commands can be combined into a single macro line. In the simplest case, this can be useful to reduce the number of lines in a long macro and to make it more readable by grouping answers that logically form an entity. The following lines, which are used to specify a matrix which is written into,

```
mf4      / specify impedance matrix
yes
upqau
auto impedances scen. %s%
yes
0
```

can be replaced by the single compound macro command

```
~+|mf4|y|upqau|auto impedances scen. %s%|y|0 /specify impedance matrix
```

In this case the character | was used as separator, but any other printable character could also be used for this purpose, as long as it does not appear anywhere else in the command.

The compound command is particularly useful after conditionals (~?...). Remember that the next line following a conditional will be executed or skipped, depending whether the condition holds or not. By using a compound statement following the conditional, the condition can be applied to several logical lines. The following example shows a test which verifies if the macro was called in module 0.00 (the main menu). If not, an error message is displayed and the macro jumps to the end:

```
~?m>000
~+|~/Error: macro must be called from main menu!|~$end
```

As seen in the example above, a compound command may contain any combination of macro commands, the only exceptions being other compound commands (~+...)or macro labels (~:...). Look at the following example, which sets the register x to the first macro parameter, if the latter has a non-zero value, or to 1 otherwise.

```
~+|~x=%1%|~?x=0|~x=1
```

Care must be taken when using substitutions within compound commands. The substitution is done once when the compound command is read as a whole. The maximum length of 128 characters applies both before and after this substitution. Just before it is executed, each sub-command is then again checked for substitutions. This is particularly important if registers are modifed within the compound command. What is the output of the following sequence?

```
~+|~x=1|~y=5
~+|~x+1|~x*%y%|~/x=%x%
```

(If you guessed the output is "x=10", you were wrong! Try it out!) If it is necessary to avoid substitution at the level of the compound statements, use %%%...%%% instead of %...%. This way, in the first substitution, the %%% are replaced by % and the actual substitution takes only place before the sub-command is executed.

Another special feature of the compound command is the possibility of branching back to the beginning of the compound command, in other words, starting it all over. This is done by executing an empty branch (without any label) as part of the compound command. This type of branching is particularly efficient, since, in contrast to branching to a label, it does not imply any file I/O. Can you see what the following "one-liner" does?

```
~+:~o|6:~*[q=%%%q%%%]:~?!e:~$:~/[error %%%e%%%]:~$
```

EXERCISE 4.1: Write a small macro which implements an empty loop counting from 1 to a number specified as the first macro parameter. Compare implementations with and without compound macro statements.

5 Text Manipulations

Ten text registers, each holding up to 124 characters, allow storing of any kind of text information. The registers `t1`–`t8` are general purpose read/write registers with no predefined special usage, while text registers `t0` and `t9` have some special properties.

Text registers are written into with the "set register" command (e.g. "`~t5=this is a text`"), and are read by means of the text register substitution (e.g. "`%t5%`"). Special variants of the substitution keys allow accessing of substrings by limiting the number of characters or skipping initial characters. Using the same example as above, "`%t5.2%`" will be replaced by the string "`th`", and "`%t5.-2%`" becomes "`is is a text`". Try to analyze the following sequence (hint: %1% contains a matrix identifier):

```
~t5=%1%
~t9=%t5.2%
~?t9=md
~t5=mo%t5.-2%
~?t9=mo
~t5=md%t5.-2%
```

As illustrated above, text registers can be compared with another text, using conditionals. The < and > comparisons are based on the standard ASCII collating sequence.

Register `t0` behaves exactly as the other text registers, but its contents is always equivalent with the current macro parameters, i.e. the string "%1% %2% %3% ...". Using `%t0%` to manipulate the entire set of macro parameters has also the advantage that even more than 9 parameters can be handled correctly. Shifting the parameters with the `~%` command will, of course, also modify the contents of `t0`. Modifying `t0`, on the other hand, also implies changing of the macro parameters. In the following sequence, the macro tests if it has been called without parameters, in which case it "installs" a set of default parameters:

```
~?t0=
~t0=ul1 1.0 yes
~/Using macro parameters: %1% %2% %3%
```

This special property of `t0` can also be used to split a given text into fields or words, by assigning the text to `t0` and accessing the words as %1%, %2%, etc. If this is done, the "real" macro parameters can be stored away into another register and reinstalled later on, as shown in the following sequence:

```
~t9=%t0%
~t0=~*Enter: First and family name=
~t1=%1%
~t2=%2%
~t0=%t9%
~/First name: %t1%     Family name:%t2%
```

The above example also shows how a user can be prompted to enter a text which is stored in a text register. This is useful in circumstances where the same parameter is to be used in different places in the macro.

Register `t9` is the only *global* register. This means that its value may be modified by a macro running at a lower level. Hence, `t9` is predestined to be used to pass return parameters back from a sub-level macro.

By the way, don't forget that text registers can of course also serve to store numerical values! This is often useful when running short of numerical registers.

EXERCISE 5.1: Write a sequence of macro commands which prompts the user with the question "`Do you really want to proceed?`" and proceeds only if the user enters a "yes", or any other string starting with the letter "y". The macro stops if a "no" (or another string starting with "n") is entered. Any other answer will cause the question to be asked again.

EXERCISE 5.2: Write a macro sequence to extract a variable substring from text register `t5`, the starting position is contained in the `x` register and the length of the substring is given in the `y` register. The resulting substring is stored in `t9`.

6 Controlling the Output with the Register O

The output control register o is a bit mapped register, which is used to control the various aspects of the dialog output during the execution of a macro. The following table lists the function associated with each bit:

Output Control Register O		
bit:	value:	description:
0	1	Suppress standard dialog output. If bit 0 is ON, the standard dialog output generated by EMME/2 is not displayed on the screen. Instead, this output is directed to a scratch file, which is automatically erased at the end of each module. Thus, activating this bit will "silence" the macro, so that only output which is generated explicitly by the macro via macro comments will be displayed on the screen. If bit 0 is OFF (default), all dialog output is directed to the screen.
1	2	Suppress comment prefix. If bit 1 is OFF (default), macro comments will be echoed at the screen including the ~/ command prefix, so that the comment is clearly identifyable as a macro comment. If bit 1 is ON, the leading ~/ characters are stripped before echoing and only the comment itself is displayed on the screen.
2	4	Suppress echo of macro input. If bit 2 is OFF (default), the answers generated by the macro are displayed on the screen (preceded by the character "<"). In addition, if switch 15, "dialog/macro debug mode", is ON, all macro commands are echoed as well on the screen. If bit 2 is ON, the output of the answers generated by the macro, as well as the macro commands, are *not echoed* on the screen as they are executed.
3	8	Macro single step tracing mode. If this bit is ON, the macro single stepping mode is activated. This mode can be used for macro debugging, as described in section 9.
4	16	Fall back to single step tracing mode. If this bit is set and a fatal error condition occurs, the macro processing is not terminated (as is done by default), but the single step mode is entered, so that the user can analyze the problem and, if necessary, take corrective action. See section 9 for further information.
5	32	Suppress programmed pauses. After certain operations, EMME/2 waits for a few seconds, in order to give the user some time to read a displayed message (e.g. after a report has been sent to the print file). When running a macro, these pauses are not always needed — sometimes they will even significantly slow down the execution of a macro. Activating bit 5 will suppress any such programmed pauses that normally happen during module execution.

Output Control Register O		*(... continued)*
bit:	value:	description:
6	64	Read an extra input line at programmed pauses (regardless of the setting of bit 5) and just before leaving a module. While the contents of the input line itself is ignored, the important aspect here is that it allows macro commands to be executed at this point. At pauses (which are usually followed by an erase of the screen), a macro may implement a user prompt to avoid that the information on the screen disappears before the user has taken note of it. At the end of a module, a macro may use this extra input line to do clean-up or post processing work before module specific information (such as the content of the g*N* registers) is lost. For these special input lines, the register q assumes the value -1.
		All following bits are automatically reset after the corresponding function has been performed.
8	256	Erase screen. When this bit is set, the screen is erased and the cursor is positioned at the top left corner of the screen.
9	512	Rewind scratch file. If this bit is set, the scratch file used to store the dialog output of the current module is rewound. This function is only useful if bit 0 was set (or still is) during the call to the current module.
10	1024	Backspace scratch file. When this bit is set, the scratch file used to store the dialog output of the current module is backspaced by one record. This function is only useful if bit 0 was set (or still is) during the call to the current module.
11	2048	Read record from scratch file into register t9. When this bit is set, the next record from the scratch file is read and stored into the text register t9. From there it can be accessed by the macro in the usual manner. This function is only useful if bit 0 was set (or still is) during the call to the current module.
12	4096	Close and discard current scratch file. If this bit is set, the current scratch file is closed and deleted. If bit 0 is still ON, a new scratch file will be opened automatically as soon as the next dialog output is generated. Activating this bit might be needed in macros which generate a great quantity of dialog output in a single module (loops!), in order to avoid disk space problems.

In order to set up a macro to be "quiet", i.e. not to display the dialog that the macro generates while running, the o register is usually set to the value 7 (=1+2+4). This corresponds to setting the bits 0, 1 and 2 to ON, implying that neither dialog output nor input is displayed and that macro comments are echoed without the leading "~/".

Since it becomes difficult to debug a macro once the dialog is hidden this way, it is

recommended to test switch 15 before setting the o register. This way, if ever needed, the macro can be made "noisy" by simply issuing the command on=15 at the main menu, without any need to edit the macro. This could be done by inserting the following sequence at the beginning of a macro:

```
~?!i&32768      / if switch 15 (dialog/macro echo mode) is OFF
~o=7            / set o-register to hide all dialog
```

If the macro should start with a screen erase (similar to an EMME/2 module), it suffices to set bit 8 of the o register. In the above example this could be done by replacing the value 7 by 263 (=7+256). If the screen is to be erased at some place within the macro, this can be done with the bitwise OR command "~o|256", which does not affect the setting of any other bits.

The bits 9–12 allow a macro to access the dialog that was sent to the scratch file instead of the screen. This way, a macro get the possibility to interpret context dependent information that is not available in any register, but appears somewhere in the dialog output.

EXERCISE 6.1: Write a macro which exports all predefined windows into an external file, which then can be used as a macro in another data bank to reinstall the same set of predefined windows.

7 System Parameter Access with Register P

The macro language provides read-only registers for the most important system parameters, such as current module, current scenario or the current date. However, when writing complex macros, it is often necessary to access all kind of other data bank or system parameters. The special register p is available for accessing such parameters. For this purpose, each available parameter is assigned an address, which is to be used to access this parameter. When the p register is set to such an address, the value of the corresponding parameter is used when the p register is used in substitutions or conditionals. Thus, the p register is different from the other registers, since it behaves differently for read and write accesses: parameter *addresses are written* to the p register, and parameter *values are read* from it.

The classes of parameters that are accessible via the p register are: global data bank parameters, scenario numbers, scenario parameters, and other system parameters. The following table lists the corresponding addresses:

System Parameter Register P	
address:	contents:
1-80	Global parameters, as described in Appendix C.1 of EMME/2 User's Manual.
101-200	External scenario numbers (0: scenario slot empty, -1: scenario slot not available in data bank)
1001-1280	Scenario parameters 1-280 of the current scenario, as described in Appendix C.1 of EMME/2 User's Manual.
2000-2011	System parameters, as defined below
2001	Software (1: EMME/2, 2: STAN)
2002	Licence number
2003	Licence size (1-12)
2004	Type of operating system (1: UNIX, 2: DOS, 3: VMS)
2005	CPU time used since creation of the data bank (in 1/10 secs)
2006	Current year (0-99)
2007	Current month (1-12)
2008	Current day of month (1-31)
2009	Current hour (0-23)
2010	Current minute (0-59)
2011	Current second (0-59)

As an example, look at the following example sequence, which computes the number of entries that are still available in the turn table of the current scenario. This number is computed as the difference of the global parameter 55 (MTURN, maximum turn table size in data bank) and the scenario parameter 4 (NTURN, current size of turn table), as follows:

```
~p=55          / global parameter MTURN, maximum size of turn table
~x=%p%         / X=MTURN
~p=1004        / scenario parameter NTURN, current size of turn table
~x-%p%         / X=X-NTURN
~/ Number of available turn table entries in scenario %s% is %x%.
```

Note that there is a potential synchronization problem when accessing the system parameters 2006 – 2011 for assembling a string with the current time. Imagine e.g. that this operation was started just a few milliseconds before midnight, so that instead of 92-05-16 23:59:59 one could obtain the string 92-05-16 00:00:00, which would clearly be wrong. To avoid such problems, the internal time values are replaced only if either more than 2 seconds passed since last time parameter access, or the last time parameter accessed was of higher or equal index. With these rules, no synchronization problems occur when accessing time parameters sequentially in the order given above.

EXERCISE 7.1: Write a macro which scans the data bank for scenarios which are prepared for assignment, but not yet assigned, and perform the corresponding assignments.

EXERCISE 7.2: Write a "wrapper" macro `cputime` which calls the macro given as parameter and, after completion, reports the number of cpu seconds that were used during the execution of that macro.

EXERCISE 7.3: Write a macro `filecopy` which copies the file given as the first parameter into the file given as second parameter. The macro should work equally well on DOS, UNIX and VAX/VMS systems.

8 Using Scalars and Floating Point Registers

Scalars have a wide-spread use throughout the entire EMME/2 system. They are accessible in the following ways:

Matrices: Scalar matrices are fully integrated into all tools of the matrix editor.

Functions and expressions: Scalars are accessible via the intrinsic function `ms()`, they can be used as "programmable" constants or to implement lookup tables.

Network: Scalars can be accessed in network calculations by means of the `ms()` intrinsic. Summary results of network calculations can be saved into scalars, including sum, average, minimum/maximum value and element, number of elements processed and number of attribute values changed.

Macros: Scalar substitution %`ms...`% can be used to insert scalar values, names and descriptions directly into dialog input and macro commands.

These possibilities make scalars the ideal vehicle to carry numeric values from one part of the system to another. Since, besides its numeric value, each scalar also has a name, description and a timestamp, they are also convenient places to store numeric results computed with a macro, along with their description.

By convention, the high-numbered macros `ms90`–`ms99` should be reserved for storing temporary values. This way, macros can freely use scalars in this range, without risking any conflict with application specific permanent scalars.

Before Release 8, the macro registers were limited to integer and text values, so that the explicit manipulation of scalars was the only means to perform floating computations within a macro. With Release 8, a set of *250 single precision floating point registers* named `r1` to `r250` is introduced. These registers can be used in the same way as integer registers: they can be set to a value (`~r15=3.14159`), operated upon (`~r15+0.5`), tested (`~?rx<0`) and substituted (%`r15.5_10`%). With these floating point registers, many tasks which before had to be implemented using scalars just to do floating point operations can now be done more easily and efficiently. ← *Rel.8!*

In all commands that operate on floating point registers, the register number can either be specified directly (`r1 ...r250`) or by using the current contents of one of the integer registers `x`, `y` or `z` (`rx`, `ry`, `rz`). With this indirection, a range of floating point registers can also be used as a vector which can easily be operated upon within index loops.

When working with floating point registers it is to be noted that, when switching from one module to the next or when invoking a lower level macro as a subroutine, their values are preserved by writing them (except those that are zero) to the external file `usemacro`. Even though this writing is done with a very high precision using the exponential format `e16.8`, the writing and reading may sometimes cause very tiny rounding errors, since there is no exact equivalence between the binary and decimal representation of fractional numbers. This does not apply to integer numbers up to 4'194'303 – these can be safely stored in floating point registers without any danger of rounding effects.

Another new enhancement that is introduced with Release 8 integrates the `get()/put()` stack of the EMME/2 expression evaluator into the macro language. The `get()/put()` stack are 250 words of auxiliary memory which are accessed from within expressions (e.g. functions, calculators) with the following intrinsic functions:

`put(`x`)`: The value of the the argument expression x is pushed on the stack and the stack pointer is incremented. The return value of the function is x.

`get(`i`)`: The value of the i-th element of the stack is returned. If i is not an integer, the truncated value is used as index.

`puti(`i`)`: The stack pointer is set the i, so that the next call to `put()` will write to the i-th element of the stack. If i is not an integer, the truncated value is used as index. ← *Rel.8!*

Note that the `get()/put()` stack is module specific, i.e. its values are set to zero when entering a new module and the final values are lost when the module is left.

Starting with Release 8 , the contents of the `get()/put()` stack can also be accessed —for both reading and writing— directly from macros. This is done using the special floating point registers `g`N. These use exactly the same syntax as the standard floating point registers `r`N, but behave somewhate differently: ← *Rel.8!*

- Any change in the `g`N register will immadiately change the return value of the corresponding calls to `get()`.
- Any call to `put()` will immediately change the value of the corresponding register `g`N.
- At the start of a new module, the registers `g`N are initialized to zero.
- When a module is terminated the values of the `g`N registers are lost. (If needeed, bit 6 of the control register `o` can be used to "collect" stack values at the end of a module.)
- The `r`N registers are *global* registers, i.e. the same values are visible from all levels of macro invocation. Hence, it is possible to use these registers to pass return values from a lower level macro back to the calling macro (in this case, of course, it is your responsibility to make sure that there are no conflicts with any `get()/put()` calls used in the same module).

The `g`N registers allow an easy implementation of many tasks that, before, were not possible or very cumbersome to carry out, such as e.g. installing lookup tables for, or collecting side results of network and matrix calculations.

EXERCISE 8.1: Write a macro `plotline` which takes a transit line name as parameter and determines the smallest network window covering the entire itinerary and then generates a plot with the transit volumes of this line.

EXERCISE 8.2: Write a macro `speedist` which generates as output a table showing the distribution of the number of car kilometers traveled per speed intervals of 5 km/h. ← *Rel.8!*

EXERCISE 8.3: Write a macro `mlookup` which contains the values of a lookup table $(f_k, k = 1, \ldots, K)$ and performs a "staircase lookup" on a given matrix u_{pq} to obtain the result $r_{pq} = f_k$, where k is chosen such that $(k-1)\delta < u_{pq} <= k\delta$, where δ is an given interval size. *Optional variant:* Replace the staircase function by a continuous piecewise linear function. ← *Rel.8!*

9 Debugging Macros

Even when writing a very simple macro, never expect it to run flawlessly on the first try. Once the first version of the macro is written, the fun part of testing and debugging starts...

Up to the Release 6, no special debugging tools were available, except for switch 15. This switch, if set to ON, will cause all macro commands to be displayed on the screen while they are being executed. So, with a bit of experience and patience, switch 15 allows a macro writer to locate about any type of problem. But the really frustrating part of this way of macro debugging is that, as soon as an error occurs, the macro processing terminates immediately. This means that the macro has to be started over and over again, until all errors are found and eliminated.

With Release 7, a macro debugging tool, called *single step tracing mode*, has been incorporated into the macro language. This new feature offers the following possibilities:

- Execute macros in a step-by-step mode, prompting the user explicitly before each macro command is executed.
- Allow inspection of all registers and change their values if needed.
- List the contents of the macro.
- Inspect current position at all macro call level.
- Take corrective ad-hoc measures, such as skipping, inserting or replacing lines.
- Allows error recovery by falling back to single stepping mode in case of error or interruption.

The single stepping mode is activated by the setting of bit 3 of the output control register o. Once activated, it will take effect at the level of each macro line which is echoed on the dialog output with "<....". Which lines exactly are echoed this way is controlled by the setting of switch 15, and bits 1 and 2 of the o register.

Each macro line that is subject to single step tracing (i.e. which is displayed on the dialog in the "<...." format) will be displayed as usual, followed by the macro corresponding line number surrounded by tilde characters. Line and character position is listed for a macro command which is part of a compound command. The user is now expected to enter a carriage return to indicate that he is now ready to process this line.

The following example, which was taken from a macro that sets the module parameters of a graphic module, illustrates the single stepping mode:

```
Enter: Width,height of nodes without node number
       (  4.00,  4.00)= < 4.0,4.0 ~L15~                          <- CR
Enter: Width,height of nodes with node numbers
       ( 32.00, 12.00)= < 32.0,12.0 ~L16:4~                      <- CR
Enter: Offset to the right for links (  1.80)= < 3.0 ~L16:15~    <- CR
```

At each ~ prompt of the single step mode, the user has the possibility to enter either a carriage return or one of the following debugging commands:

Debugging Commands	
command:	action:
*	Display the current values all integer registers.
t*	Display the current values all text registers.
r*	Display the values all non-zero general purpose floating point registers.
g*	Display the current values all non-zero elements of the get()/put() stack, i.e. the g*N* registers.
reg	Display current value of the given register *reg*.
reg=value	Set register *reg* to the specified value.
:c	Continue macro - leave single stepping mode. Bit 3 of the o register is reset and bit 4 (fall-back) is set, so that the single step mode is reactivated again in case of error or interrupt.
:l	List current context in the macro (from 3 lines before to 3 lines after the current line).
:l*N*	List *N* lines of the macro starting with the current line.
:l-*N*	List *N* lines of the macro ending with the current line.
:p	Display macro position at all macro call levels.
:r*text*	Replace the current line by *text* before executing it.
:s	Skip the current line (do not execute it) and proceed to the next line. (Not valid in compound commands.)
:i*text*	Insert and execute the line defined by *text* before executing the current line. (Not valid in compound commands.)
:x	Exit macro processing.
?	Display the list of available debugging commands.

Any text that follows a debugging command is subject to the usual macro substitution rules, i.e. %...% sequences will be substituted as if they would appear in the macro itself.

As indicated above, bit 4 of the output control register o activates the fall-back mode. In this mode of operation, the macro executes normally until an error condition or a user interrupt is detected. By default these events would terminate the macro processing immediately with the messages >ERROR...> or >INTERRUPT>. In fall-back mode the macro enters single stepping mode, so that the problem can immediately be analyzed and, if possible, corrected.

10 Recommendations for Trouble-Free Macro Writing

We conclude this tutorial with a list of recommendations which should help to write robust and transportable macros that will perform well in different applications, and —last but not least!— will avoid unnecessary problems when switching to new releases:

- When a macro is leaving a module, always use "`quit`" at the level of the primary select. Do not use the explicit "*n*=`end`" option, since this would require explicit modification if in a later release new options are added to the module.
- Do not assume that all users and all applications use the same switch and module parameter settings as you do. Whenever possible, write macros to be independent of particular switch or module parameter settings. Often this can be done very easily by inserting conditionals which test the switch setting directly (`~?i&...`) or that verify the presence of an optional question (`~?q...`).
- When a macro needs to modify switches, make sure that they are set back to their initial value at the end of the macro. Saving the initial state of all switches is done easily by saving the value of the switch register `i` at the beginning of the macro. This allows, at the end of the macro, to reset those switches that were changed in the macro to their initial state.
- It is a good practice that macros which generate reports, plots and batch output files use the flexible file naming commands `reports=`, `plots=` and `batchout=` to define their own file names. However, if these commands are used it is important to reset the modified file names to the setting that were active before the macro was started. This can be achieved by using the commands `reports=^`, `plots=^` and/or `batchout=^`, which will set the file names back to the names that were active before the last change.
- External commands (`~!...`) are executed by the operating system of the host computer. Such commands may render the macro incompatible with different host computers. If a macro contains such dependencies, this should be clearly documented. Better even, the macro should test for the correct operating system (using system parameter 2004), or, if possible, adapt itself to the various host types by providing several versions of the `~!` command, each preceded by the corresponding conditional.
- It is recommended that macros which modify the data bank should create a log book comment when they are started. If this comment contains the macro name and the run time parameters (e.g. `c='mymacro %1% %2% %3%'`), the log book can later be consulted to reconstruct the history of the data bank. For macros that run for a long time and have a high risk of being interrupted prematurely, it is sometimes even advisable to generate a log book entry at the beginning and one after normal completion, so that runs that did not terminate normally will be recognizable as such in the log book.
- If a macro requires a certain number of parameters, a test should be done at the beginning to verify if enough parameters are provided. If not, a message should

be displayed which shows a summary of the usage of the macro. This way, a user which is not sure about the calling convention of a macro, can always call it without parameters to obtain instructions on the calling sequence.

- Many macros need to store auxiliary data vectors that are just needed during the macro run, but do not have to be preserved. If these vectors are limited to use within the network calculator, the best choice is to use the temporary network attributes `tmp`*xy*, as these are always available and will be deleted automatically when module 2.41 is left. If the auxiliary vectors are used outside the network calculator, they should either be specifyable as macro parameter, or be stored in extra attributes named `@tmp..` that are created at the beginning and deleted at the end of the macro.

- Scalars in the range `ms90` – `ms99` should stay reserved for use as scratch pads. By observing this rule, macros which need scalars to hold temporary data, can always use these scalars without any danger that this would distroy any important data.

- When using conditionals on floating point registers, be very careful with testing for equality. Unless the contents of the floating point register is known to be a true integer (i.e. stored there as integer value or obtained by operations involving only integer values) in the range 1 to 4194303, rounding errors may occur that make the use of strict equality comparisons unsafe.

- Switch 15 is a convenient way of activating the output of internal debugging information. This switch can safely be assumed OFF during normal operation. If the macro finds it to be ON, this could be used to set the output control register to echo dialog output and input and activate the fall-back to single step mode. If the macro creates auxiliary data vectors which are usually deleted at the end, it is sometimes useful not to delete them when switch 15 is ON, to allow analyzing them post-mortem.

11 Solutions to the Exercises

This section normally contains the solution macros for all exercises in the previous sections.

But as said in the introduction, the pupose of the exercises is of course to solve them yourself. Therefore, handing out the solutions here along with the tutorial would (for all but the most disciplined among us!) spoil all the pains and pleasures of working out the solutions yourself.

For this reason, when teaching this course, I only hand out the solutions at the very end of the course. If you downloaded this tutorial from my Web site and have worked through it on your own, you can send me an email at `heinz@spiess.ch` or mail me a diskette in which you include your solutions (or however far you got with them). I will look at them and then send you back my set of solutions.

Of course, also if you have any question regarding this tutorial in general or some of the exercises in particular, please feel free to contact me and I will try my best to reply to your queries. But note that this is not part of my official EMME/2 support activities which I do on behalf of INRO. Therefore, please be patient if I do not reply to you with the same fast turn around time that you are used to from sending me EMME/2 software support requests.

Heinz Spiess, EMME/2 Support Center

Address:	Haldenstrasse 16 CH-2558 Aegerten Switzerland
Phone:	++41 32 53 20 57 *(number will change to ++41 32 373 20 57 on Nov 9, 1996!)*
Fax:	++41 32 53 57 86 *(number will change to ++41 32 373 57 86 on Nov 9, 1996!)*
Email:	`heinz@spiess.ch`
WWW:	`http://www.spiess.ch/emme2`

Exercise 2.1: (File: resmat.mac)

```
01: %1%        / matrix identifier       (Enter: Matrix ( mf )=)
02: y          / initialize matrix if it doesn't exist, change header if it does
03: upqau      / matrix name             (Enter: Name (6 char)=)
04: 'auto impedance for scenario %s%'  / (Enter: Matrix description=)
05: ~x=%q%     / preserve current value of q (why?? this is the trickiest part!)
06: ~?x=0      / matrix is being created
07: 0          / default value           (Enter: Default value=)
08: ~?x=1      / matrix exists already
09: no         / don't initialize values (Initialize data?)
```

Exercise 2.2: (File: testrel.mac)

```
01: ~x=%v%         / compute major release level of current module
02: ~x/100
03: ~?x=7          / test if software is on release level 7
04: ~$release_is_okay
05: ~/ This macro is written for release 7, you are using release %x%!
06: ~/ Ask the author of this macro to provide you with an updated version.
07: ~$end_of_macro
08: ~:release_is_okay
```

Exercise 3.1: (File: inireg.mac)

```
01: ~/ inireg - wrapper to initialize all registers before calling a macro
02: ~+|~x=0|~y=0|~z=0|~p=0|~o=0|~t1=|~t2=|~t3=|~t4=|~t5=|~t6=|~t7=|~t8=|~t9=
03: ~?v>800  / initialize floating point registers for Release 8 and later
04: ~+|~x+1|~rx=0|~?x<250|~$|~x=0   /   fast loop over all 250 rN registers
05: ~<%t0%
```

Exercise 3.2: (File: factor.mac)

```
01: ~x=%1%         / compute factorial %1%!
02: ~t9=1
03: ~?x<2          / result is trivial for 0 and 1
04: ~$return
05: ~x-1           / compute factorial (%1%-1)! by recursion
06: ~<factor %x%
07: ~x=%t9%        / result value is in t9
08: ~x*%1%         / %1%! = %1%*(%1%-1)!
09: ~t9=%x%
10: ~:return
11: ~/ %1%!=%t9%
```

Exercise 4.1a: (File: slowloop.mac)

```
01: ~x=1
02: ~/starting loop from 1 to %1% (using label branching)
03: ~:loop
04: ~x+1
05: ~?!x>%1%
06: ~$loop
07: ~/loop from 1 to %1% done!
```

Exercise 4.1b: (File: fastloop.mac)

```
01: ~x=1
02: ~/starting loop from 1 to %1% (using compound command)
03: ~+|~x+1|~?!x>%1%|~$
04: ~/loop from 1 to %1% done!
```

Exercise 5.1: (File: confirm.mac)

```
01: ~:confirm
02: ~t9=~*Do you really want to proceed?
03: ~t9=%t9.1%
04: ~?t9=n
05: ~$end_of_macro
06: ~?!t9=y
07: ~$confirm
```

Exercise 5.2: (File: substr.mac)

```
01: ~/ Construct variable substring of t5 starting at position x with length y.
02: ~x-1                / number of characters to be skipped from the beginning
03: ~+|~t9=%%%t5.-%x%%%%|~t9=%%%t9.%y%%%%
04: ~x+1                / restore initial value of x, in case it is still needed
05: ~/ substr(t5,%x%,%y%)="%t9%"
```

Exercise 6.1: (File: getwindw.mac)

```
01: ~o=7
02: ~/*************** getwindw (DOS and UNIX only) ****************
03: ~/Saving the following predefined windows into file putwindw.mac:
04: ~+|2.13|1|n|~?q=1|n|all|y|4  / get to window dialog in 2.13
05:  ?                          / list predefined windows to scratch file
```

```
06: ~o|1024                        / backspace over 'Enter: Window (a-z)='
07: ~!echo ~/predefined windows saved by getwindw  on %d% >putwindw.mac
08: ~:loop
09: ~o|1024                        / backspace over last line
10: ~o|3072                        / backspace and read a line
11: ~?t9> W                        / is it the table header?(' WINDOW....')
12: ~$done
13: ~/ %t9%
14: ~!echo %t9% >>putwindw.mac
15: ~$loop
16: ~:done
17:                                 / provoke error to get back to select
18: ~?e                             / catch error condition
19: q                               / quit module
20: ~o=6                            / Reactivate dialog output
21:                                 / redisplay 'Enter: Next module='
```

Exercise 7.1:

(File: assign.mac)

```
01: ~o=7       / ASSIGN:  perform all prepared auto and transit assignments
02: ~?i&32768  / if switch 15 (dialog/macro echo mode) is ON
03: ~o=8       / enable dialog output and set single stepping mode
04: ~/ Scan all scenarios and perform any assignment which has been prepared
05: reports='assign.rep'   / define file name for reports
06: ~t1=%s%    / store current scenario in t1
07: ~y=0       / initialize counter for auto assignments
08: ~z=0       / initialize counter for transit assignments
09: ~p=100     / set up counter x for stepping through scenario numbers
10: ~:next_scenario
11: ~p+1       / increment scenario counter
12: ~?p=0      / zero scenario number - scenario is not used
13: ~$next_scenario
14: ~?p<0      / negative value indicates that all scenarios have been scanned
15: ~$done
16: s=%p%      / scenario exists - define it as current
17: ~?f&32     / scenario ready for auto assignment?  If yes, assign it
18: ~+|5.21|~/ Auto assignment scenario %s%    (do not interrupt!)|~?q=2|2|~y+1
19: ~?f&64     / scenario ready for transit assignment?  If yes, assign it
20: ~+|5.31|~/ Transit assignment scenario %s% (do not interrupt!)|~?q=2|2|~z+1
21: ~$next_scenario
22: ~:done
23: s=%t1%     / reset scenario to what it was initially
24: reports='^'/set report file back to original setting
25: ~/ ASSIGN terminated normally after %y% auto and %z% transit assignments.
26: ~o=6       / reset dialog output
27:            / display 'Enter: Next module='
```

Exercise 7.2:

(File: cputime.mac)

```
01: ~p=2005  / cputime - wrapper macro to report CPU time used by any macro
02: ~x=-%p%  / usage:  ~<cputime macroname par1 par2 ...
03: ~<%t0%
04: ~x+%p%   / compute difference of CPU time
05: ~y=%x%   / CPU times are stored in 1/10th secs
06: ~y%10    / store fractional part in y
07: ~x/10    / store integer part in x
08: ~/ CPU time used by macro %1%: %x%.%y% secs
```

Exercise 7.3:

(File: `filecopy.mac`)

```
01: ~x=%0%        / test if exactly two parameters are specified
02: ~?!x=2
03: ~+|~/ usage: filecopy <fromfile> <tofile>|~$end
04: ~p=2004
05: ~?p=1         / UNIX
06: ~!cp %1% %2%
07: ~?p=2         / DOS
08: ~!copy %1% %2%
09: ~?p=3         / VAX/VMS
10: ~!COPY/LOG %1% %2%
11: ~:end
```

Exercise 8.1a:

(File: `plotline.mac`)

```
01: ~o=7
02: ~/ Plotline:  Plot the transit volumes of the lines given in parameters
03: ~/            by using an ‘‘optimal’’  window which is computed first
04: ~/            (using scalars)
05: ~/ ... computing optimal window for transit line %1%...
06: ~?i&32768  / if switch 15 (dialog/macro echo mode) is ON
07: ~o=8       / enable dialog output and set single stepping mode
08: 2.41       / use network calculator to compute min/max coordinates
09: 1          / network calculation for x coordinate
10: n          / no results saved
11: 'xi+0*hdw' / x-coordinate of i-node of segment  (note trick with 0*hdw!)
12:
13: %1%        / selected line(s) only
14:
15: all        / all links
16: 5          / save summary results in scalars
17: 2          / minimum
18: ~?q>0      / no lines selected? (invalid input)
19: ~+|~/ Line %1% does not exist!|q|~$end
20: ms96
21: lowX       / scalar name and description
22: 'lower X coordinate for line %1%'
23:            / don't save line index
```

```
24:            / don't save i-node
25:            / don't save j-node
26: 3          / maximum
27: ms97
28: uppX       / scalar name and description
29: 'upper X coordinate for line %1%'
30:            / don't save line index
31:            / don't save i-node
32:            / don't save j-node
33: r          / back to primary select
34: 1          / network calculation for y coordinate
35: n          / no results saved
36: 'yi+0*hdw' / y-coordinate of i-node of segment  (note trick with 0*hdw!)
37:
38: %1%        / selected line(s) only
39:
40: all        / all links
41: 5          / save summary results in scalars
42: 2          / minimum
43: ms98
44: lowy       / scalar name and description
45: 'lower y coordinate for line %1%'
46:            / don't save line index
47:            / don't save i-node
48:            / don't save j-node
49: 3          / maximum
50: ms99
51: uppy       / scalar name and description
52: 'upper y coordinate for line %1%'
53:            / don't save line index
54:            / don't save i-node
55:            / don't save j-node
56: q
57: ~/ ... plotting volumes of transit line %1%...
58: 6.22       / now plot the selected transit line or lines
59: 1          / transit volumes
60: %1%        / selected line(s) only
61:
62: 1          / bars only
63: n          / no color index
64: n          / no node numbers
65: ~?q=1
66: n          / no base network
67: all        / all links
68: y          / window
69: 1          / using coordinates
70: %ms96% %ms98%  / lower left corner
71: %ms97% %ms99%  / upper right corner
72: ~:end
73: ~o=6        / enable dialog output
```

```
74:
```

Exercise 8.1b: (File: plotlin2.mac)

```
01: ~o=7
02: ~/ Plotlin2:  Plot the transit volumes of the lines given in parameters
03: ~/            by using an ‘‘optimal’’  window which is computed first
04: ~/            (using floating point registers)
05: ~?v<706    / test for Release 8 sice macro uses floating point registers
06: ~+|~/ This macro requires EMME/2 Release 8 or later!|~$end
07: ~/ ... computing optimal window for transit line %1%...
08: ~?i&32768  / if switch 15 (dialog/macro echo mode) is ON
09: ~o=8       / enable dialog output and set single stepping mode
10: 2.41       / use network calculator to compute min/max coordinates
11: 1          / network calculation for x coordinate
12: n          / no results saved
13: ~+|~g1=-9999999|~g2=9999999|~g3=-9999999|~g4=9999999  / initialize registers
14: '0*hdw+'   / dummy term to force a segment expression
15: put((put(get(1).max.xi.max.xj)-put(get(2).min.xi.min.xj))
16: .max.(put(get(3).max.yi.max.yj)-put(get(4).min.yi.min.yj)))
17:
18: %1%        / selected line(s) only
19:
20: all        / all links
21: 5          / no report, no punch
22: ~+|~r1=%g1%|~r2=%g2%|~r3=%g3%|~r4=%g4% / copy to floating point registers fN
23: ~?g5=0
24: ~+|~/ Empty window - check line names! |q|~$end
25: ~g5*.03    / compute border increment
26: ~+|~r1+%g5%|~r2-%g5%|~r3+%g5%|~r4-%g5% / adjust for border
27: q
28: ~/ ... plotting volumes of transit line %1%...
29: 6.22       / now plot the selected transit line or lines
30: 1          / transit volumes
31: %1%        / selected line(s) only
32:
33: 1          / bars only
34: n          / no color index
35: n          / no node numbers
36: ~?q=1
37: n          / no base network
38: all        / all links
39: y          / window
40: 1          / using coordinates
41: %r2% %r4%  / lower right corner
42: %r1% %r3%  / upper left corner
43: ~:end
44: ~o=6        / enable dialog output
45:
```

Exercise 8.2:

(File: speedist.mac)

```
01: ~o=7 /#### COMPUTE DISTRIBUTION OF VEHICLE KILOMETERS BY 5KM/H SPEED CLASSES
02: ~?!m=000  / test for correct starting module
03: ~+|~/This macro must be started from the main menu!|~$end
04: ~?v<706  / test for correct software version, since new features are used
05: ~+|~/This macro requires EMME/2 Release 8 Beta or later!|~$end
06: ~?!f&1024/ test if current scenario has a valid auto assignment
07: ~+|~/Scenario %s% does not have an auto assignment!|~$end
08: 2.41     / compute distribution of vehicle kilometers by 5km/h speed classes
09: 1        / network calculation
10: n        / without saving the result
11: put(volau*len+get(puti((60*len/timau)/5+1)))
12:
13: ~+|not volau=0| / for all links
14: 5        / nor report, no batch output
15: ~/---------------------------
16: ~/Vehicle kilometers by speed:
17: ~/---------------------------
18: ~x=1     / initiaize index
19: ~r1=0    / initialize lower limit of interval
20: ~r3=0    / initialize grand total
21: ~:next
22: ~r2=%r1% / compute upper limit of interval
23: ~r2+5
24: ~?gx>0   / only output non-empty intervals
25: ~/%r1_3% - %r2_3% km/h: %gx.2_12%
26: ~r1=%r2% / prepare for next interval
27: ~r3+%gx% / update grand total
28: ~x+1     / increment index
29: ~?x<250  / and loop
30: ~$next
31: ~/---------------------------
32: ~/         Total: %r3.2_12%
33: ~/===========================
34: q
35: ~:end
36: ~o=6
37: ~?m=000
38:
```

Exercise 8.3:

(File: mlookup.mac)

```
01: ~o=7
02: ~/********** MLOOKUP - Apply a lookup table to a matrix ****************
03: ~/
04: ~/calling sequence:   mlookup <upq> <rpq>
```

```
05: ~/where:              <upq>  input matrix (or expression)
06: ~/                    <rpq>  result matrix identifier
07: ~/
08: ~+|~x=%0%|~?x<2|~$end
09: ~?v<706
10: ~+|~/ This macro requires EMME/2 Release 8 or later!|~$end
11: ~+|~?i&32768|~o=8  / if switch 15 is set, go into single stepping mode
12: 3.21      / call matrix calculator
13: ~r1=2     / <- interval size
14: ~: start of lookup table
15: ~g1=.0496
16: ~g2=.0591
17: ~g3=.1021
18: ~g4=.1791
19: ~g5=.2505
20: ~g6=.3242
21: ~g7=.4041
22: ~g8=.4919
23: ~g9=.5826
24: ~g10=.6623
25: ~g11=.7255
26: ~g12=.7813
27: ~g13=.8443
28: ~g14=.8939
29: ~g15=.9318
30: ~g16=.9575
31: ~g17=.9856
32: ~g18=.9916
33: ~g19=.9952
34: ~g20=.9977
35: ~g21=.9992
36: ~g22=.9993
37: ~g23=.9997
38: ~g24=.9999
39: ~g25=1.0000
40: ~:end of lookup table
41: 1     / matrix calculation
42: 'y'   / save results in matrix %2% (see exercise 2.1)
43: ~+|%2%|y|lookup|'result of macro "mlookup %1%"'|~x=%%%q%%%|~?x=0|0|~?x=1|n
44: get(int((%1%)/%r1%+1).max.1.min.25)
45:
46:       / no constraint matrix
47: 'no'  / no submatrix
48: ~?q=2
49: 2     / send report to printer
50: q
51: ~/
52: ~/Macro "mlookup %1% %2%" terminated normally.
53: ~/calling sequence:   mlookup <upq> <rpq>
54: ~:end
```

```
55: ~/*************************************************************************
56: ~o=6
57:
```