Embedded Macro Processors

P. J. Brown
Computing Laboratory, The University, Canterbury, Kent CT2 7NF, UK

Macro processors are often found embedded in systems and applications software: assemblers, compilers, text formatters, circuit design packages, etc. These embedded macro processors are all different from one another, mainly because each has unique requirements for communicating with its host software. This paper presents a generalized off-the-shelf component, able to serve as the embedded macro processor in almost any desired host software. The key to the generality is that macros are defined using an existing high-level language such as Pascal, thus giving the writer of macros all the power of that language.

Sometimes macro processors are used as preprocessors, which run independently of any other software, and sometimes they are embedded in a particular piece of software. Examples of the latter are

(a) macro processors that form part of a macro assembler
(b) macro processors embedded in certain compilers
(c) macro processors embedded in text formatters such as **nroff**.[1]

There are three main advantages of embedding a macro processor in a given piece of *host software*. They are

(a) *Closer interaction.* The macro processor may be able to examine variables and data structures within the host software and tailor its macro replacement accordingly. Thus, for example, a macro in **nroff** can examine the current page number—to see if it is even or odd, say—when it is generating page headings. Similarly a macro processor within an assembler may be able to examine, and perhaps even to change, the assembler's symbol table.
(b) *Better syntactic control.* The host software is able to confine macro replacement to certain contexts, and the syntax of macro arguments may be made to fit the syntactic constructs of the host software. Thus in **nroff** (which has an extremely simple syntax) macros are only recognized in the context of **nroff** commands, and macro arguments are split up in the same way as arguments to **nroff** commands. A similar scheme is adopted by most macro-assemblers.
(c) *Better error messages.* A notorious problem with macro preprocessors arises as follows: the macro processor converts a source text into a program in a language L, and then passes this program to the compiler for the language L. If the compiler finds errors in the program, these come out in terms of the converted text rather than the original source text. An error in the X statement in line 100 of the converted text may really be caused by an error in the Y statement in line 40 of the original source text, and the puzzled user is normally left to relate the two. With embedded macro processors such problems can be lessened or even eliminated.

Given these three solid advantages, embedded macro processors, although each tied to one application, dominate the field, and general-purpose macro preprocessors are used relatively less.[2]

IMPLEMENTING EMBEDDED MACRO PROCESSORS

It is probably true that, although hundreds of embedded macro processors exist, no two are the same: instead each has been separately designed and implemented. This represents an enormous waste of time both for the implementors and for the macro writers, who have to learn a new (and often formidable) notation each time they use a new macro processor.

An ideal is that all embedded macro processors should be the same, or at least should share a common kernel. However such an ideal has not been achieved in any other software field, so it is not reasonable to expect it to happen with embedded macro processors. 'Not invented here' is a natural human feeling. It is, nevertheless, reasonable to expect the sort of commonality that arises with, say, programming languages: that there be half a dozen popular models, all widely used and implemented. In programming languages the top six take perhaps 90% of the market, the other 10% being shared by countless thousands of other languages. There are, of course, problems of incompatibility between different implementations of the six popular languages, but nevertheless the degree of commonality that exists has saved both users and implementors immense amounts of time.

The main reason that no such commonality occurs with embedded macro processors results from what we quoted as the first advantage of embedded macro processors: closer interaction with the host software. For example a macro assembler usually provides a mechanism for examining a symbol table, and a macro processor embedded in a text formatter must provide a mechanism for accessing page numbers, fount names, distance of indentation and so on.

Implementors of the host software find that no existing macro processor meets their special needs, and thus need no further excuse to create a new one of their own.

A POTENTIAL SOLUTION

The object of this paper is to present a potential solution whereby an existing macro processor can be used, and re-used, to serve as an embedded macro processor in diverse environments. We assert that a suitable macro

CCC-0010-4620/84/0027-0348 $03.00

processor for this purpose is SUPERMAC,[3] because it allows users *to write macros in an existing high-level language*.

Before pursuing this point we shall explain the fundamentals of SUPERMAC. SUPERMAC is a library of macro processing routines, in a similar way to NAG being a library of numerical algorithms. Like NAG, SUPERMAC is available in various high-level languages. The different implementations have common functions but a different user interface—the interface for the Pascal user is different from that for the FORTRAN user.

In this paper we shall concentrate on SUPERMAC-Pascal,[4] the implementation of SUPERMAC in Pascal. In SUPERMAC-Pascal the macro writer constructs a Pascal program that defines his macros and how they are to be replaced. This program, which will use the SUPERMAC library, is compiled in the normal way and then executed in order to apply the macros to some given source text and produce the corresponding output. Macros are therefore completely separate from the text they operate on; this contrasts with macro schemes where macro definitions are placed on the front of the text they are to process.

A SUPERMAC-Pascal macro can be defined using the library procedure *Macrop*. The first argument to *Macrop* is a string that represents the pattern of the macro to be recognized; within this pattern a pair of minus signs represents an argument. The second argument to *Macrop* is the name of a procedure in the user's program. This procedure is called every time the macro pattern is matched, and defines how the macro is to be replaced. A simple call of *Macrop* is

Macrop ('SECTION – – : – – ; ', *dosection*);

When all the macros have been defined, the SUPERMAC procedure *Mscanf* is called to process a source file, e.g.

Mscanf(*myfile*);

SUPERMAC then scans *myfile*, looking for occurrences of any of the macro patterns that have been defined. Text that does not match any pattern is copied directly to the output file; text that does match a pattern causes a call of the procedure corresponding to the pattern, and this procedure defines what text (if any) is to replace the pattern. As an example of such a procedure call, assume that the source file contains the text

SECTION A.1: Fundamentals;

When this text is encountered in the source file it matches the pattern we have defined above, and thus the procedure *dosection* is called. The arguments to the macro call—accessed via *Marg* (see below)—are the strings 'A.1' and 'Fundamentals'.

In order to show how a procedure such as *dosection* is encoded, we shall assume that SECTION is a macro for a text formatter and is such that

SECTION A.1: Fundamentals;

is mapped into

```
/space 4
/Bold
A.1          Fundamentals
/Roman
/space 1
```

The SUPERMAC library provides procedures that can be used to output text that is to form part of a macro replacement. Among these procedures are

Mstr(*S*)	output the string *S*
Mln	output a newline character
Marg(*N*)	output the *N*th argument of the current macro
Mspace(*N*)	output *N* spaces

Using these, the *dosection* procedure can be encoded as

```
procedure dosection;
begin
  Mstr ('/space 4'); Mln;
  Mstr ('/Bold'); Mln;
  Marg(1); Mspace(6); Marg(2); Mln;
  Mstr ('/Roman'); Mln;
  Mstr ('/space 1'); Mln
end; {dosection}
```

As it stands this procedure performs a straight replacement that could be done by a preprocessing macro. We shall, however, soon show macros that are dependent on their environment.

USING SUPERMAC-PASCAL

We have used Pascal as an example in this paper because Pascal is a widely known high-level language. In one sense, however, Pascal is a bad example: standard Pascal has no facility for separate compilation. Fortunately this is more a problem in theory than in practice, and almost all production Pascal systems do have a facility for separate compilation. We shall therefore assume that this is the case.

Normally the SUPERMAC-Pascal library is precompiled, and a user's program that calls the library is compiled separately and then linked with the precompiled library.

THE ADVANTAGE OF AN EXISTING HIGH-LEVEL LANGUAGE

Having explained the nature of SUPERMAC, we shall now return to our claim that it is suitable to act as an embedded macro processor. The reason is that the writer of a SUPERMAC macro has all the facilities of a high-level language at his command. These should be adequate to meet all reasonable needs (though inevitably each high-level language has its blind spots). As a concrete example, assume we want to extend our *dosection* procedure so that all the SECTION names are remembered in a table, which, can, at the end of a run, be output as a table of contents. This presents no problem: Pascal offers adequate tools (arrays, records, perhaps pointers) to build such a table, and these can be used in the *dosection* procedure. (Remember that *dosection* is an ordinary Pascal procedure; there are no special restrictions upon it just because it happens to define the replacement of a macro.)

As a second example, assume that 'major' SECTIONs—those whose names contain one number (e.g. SECTION A.1)—are to be replaced in a different way

from 'minor' SECTIONs—those whose names contain more than one number (e.g. SECTION A.1.1). This again presents no problem. Pascal **if** statements can be used to examine the form of the string that is the first argument to SECTION, and to output different replacements depending on the format of this string.

Thus SUPERMAC-Pascal provides the necessary power, assuming that Pascal itself does. It also, as we shall see, provides the necessary communication facilities to allow communication between the host software, SUPERMAC itself, and the macro writer's procedures. Communication is simply performed via global variables, procedures, and data structures, all shared among the above three components.

A PROTOTYPE SYSTEM

To illustrate this, and to demonstrate the feasibility of using SUPERMAC as an embedded macro processor, we built a prototype system.

The essence of good prototype design[5] is to produce a system quickly and cheaply that can be used to investigate the matters of interest. In our case it would have been nice to construct an elaborate piece of software such as a text formatter. However, since such programs require years of implementation effort, we settled for a much more limited piece of software, which, it is hoped, encapsulates all the necessary properties of a text formatter as regards macro expansion.

The prototype host software, which is called *Liner*, is merely a tool for adding line-numbers to a document. The input to Liner consists of lines of text with interspersed Liner commands, and the output is a listing of the input text lines with a line-number against each line. The command lines are identified by a *trigger* character at the start of the line; the initial setting of the trigger is '/'. The commands themselves control the numbering system and the format in which line-numbers are displayed. The commands we shall use in our examples are

/l *N* set the line-number to *N*
/i *N* set the line-number increment to *N* (default 1)
/w *N* set the print width to *N*
/t *C* set the trigger character to *C*
/h highlight the line that follows by printing asterisks as its line-number

As an example, if Liner is fed the following input:

```
A line
/l 100
A second line
A third line
/h
A fourth line
A fifth line
```

then it produces the output:

```
     1: A line
   100: A second line
   101: A third line
******: A fourth line
   103: A fifth line
```

As can be seen, Liner is very much a prototype to test ideas, rather than the software tool that the world has been yearning for. Nevertheless it has a fundamental similarity to a valuable tool, a text formatter.

Liner has been implemented with SUPERMAC-Pascal as an embedded macro facility. Macros are allowed anywhere within the command lines, i.e. the lines that start with the trigger character. Liner calls SUPERMAC to process each such line and then it processes the result, which is usually a sequence of one or more Liner commands. (Thus macros can be used to redefine existing commands, since Liner does not examine commands until after SUPERMAC has processed them.)

MODIFYING EXISTING SOFTWARE

An alternative to the prototype approach would have been to take an existing text formatter with embedded macro processor and to replace the latter with SUPERMAC. The prototype approach was, however, felt to be cleaner and less constrained. Moreover, this reflects the real potential use of SUPERMAC: it is unlikely that people will want to modify existing software where an alternative macro processor has already been implemented; instead SUPERMAC's likely role is with newly-built software.

IMPLEMENTATION OF LINER

Liner has been written as an ordinary Pascal program, using the SUPERMAC library. In order to illustrate built-in macros it has been given one such macro: R is a macro command to do a 'reset' and is mapped into

```
/l 1
/i 1
```

It therefore sets the line-number and increment back to their default initial values of 1. (In this paper we use macro names made up of capital letters in order to distinguish macros from built-in commands.)

Liner has been implemented in Berkeley Pascal, and runs under the UNIX† system. In Berkeley Pascal, a program references the SUPERMAC library by including the line

 # include # "supermac.h"

The file *supermac.h* contains specifications of all the procedures (e.g. *Macrop*, *Mstr*) in the SUPERMAC library, and also a number of public constants, types and variables.

In Berkeley Pascal, a procedure or function is shared between several separately compiled modules if it is declared as *external* in all of them (where *external* is used like the Pascal keyword *forward*). One of the modules must contain a declaration of the body of the procedure or function. This mechanism can be used, as we shall see, to allow the incorporation of user-defined macros as well as the built-in ones.

The overall layout of the encoding of Liner is shown in Fig. 1.

† UNIX is a Trademark of Bell Laboratories.

```
program Liner . . .;

# include "supermac.h"

procedure definemacros; external; {user-defined macros; see main text}

procedure process;

{ This is the guts of Liner. It is a procedure to read lines of the source
text and process them. If a line is a command (i.e. begins with the
trigger) it is fed to SUPERMAC, and the resultant output, which may
consist of a mixture of command lines and text lines, is then re-
processed; lines that are not command lines are listed with a prefixed
line-number.

}

procedure Rproc; {built-in macro R}
begin {output two Liner commands}
  Mstr('l  1'); Mln;
  Mstr('/i  1')
end; {Rproc}

begin {main program}
{ Initialize SUPERMAC }
  Macrop('R #', Rproc); {define built-in macro}
  definemacros; {define user macros}
  process
end.
```

Figure 1. The encoding of Liner.

The whole purpose of a macro facility is to allow the user to extend Liner; thus the system must cater for user-defined macros. This is done by calling an external procedure called *definemacros*. Each user supplies his own version of the procedure—if no such procedure is found a null procedure is assumed, i.e. no user-defined macros are present.

Liner is pre-compiled into loader format. The user of Liner either writes his own version of the *definemacros* procedures or alternatively selects a macro library that consists of a version of *definemacros* provided by the system or by another user. He then compiles this and links it with the pre-compiled versions of Liner and the SUPERMAC library in order to get a compiled version of Liner with his own macros incorporated.

Obviously the mechanics of linking and so on should be hidden from the user. In UNIX, this is achieved by a shell script called *liner*, which does any necessary compilation and linking and commences execution.

INTERACTION WITH ENVIRONMENT

We have now shown that our prototype

- is built using an existing macro processor
- provides the necessary power for writing macros
- allows for built-in macros, macro libraries and user-defined macros.

We shall now cover the interaction between macros and Liner itself.

The objects (constants, data types, variables, procedures and functions) within Liner which might be needed by macros are made public. Most of these public objects are variables. Examples are

```
linenumber: integer;   {the current line-number}
increment: integer;    {the current increment}
printwidth: integer;   {the current printwidth}
trigger: char;         {the current trigger character}
```

(In an ideal world there would be control over whether the outside user could only read these variables or could both read them and write them. Pascal does not provide this control and all variables are effectively read/write; more recent languages such as ADA and Modula-2 do provide the necessary control.)

Definitions of all the public objects in Liner are collected into a file. This file is then combined with the *supermac.h* file (which contains the definitions of the public objects in SUPERMAC) to form a declarations file called *liner.h*. A user who includes *liner.h* then has access to all the public objects in both SUPERMAC and Liner.

In order to illustrate the communication between user-defined macros and Liner we shall show a sample user-defined macro which performs a task that could not be done by a macro preprocessor. The task is to provide two macros SAVE and RESTORE: the SAVE macro saves the current values of the line-number and increment, and the RESTORE macro restores them to their previously saved values.

The definitions of these macros are shown in Fig. 2.

```
# include "liner.h"

var
  havesaved: Boolean;   {true if something saved}
  saveline:  integer;   {saved linenumber}
  saveincr:  integer;   {saved increment}

procedure SAVEproc;
begin
  saveline := linenumber;
  saveincr := increment;
  havesaved := true
end; {SAVEproc}

procedure RESTOREproc;
begin
  if not havesaved then begin {error case}
    writeln(Merr,
      'ERROR:  wrong RESTORE after input line', Mli: 4);
    writeln(Merr,
      '# last output line was numbered', linenumber: printwidth)
  end else begin
    linenumber := saveline;
    increment := saveincr
  end
end; {RESTOREproc}

procedure definemacros;
begin
  havesaved := false;
  Macrop('SAVE', SAVEproc);
  Macrop('RESTORE', RESTOREproc)
end; {definemacros}
```

Figure 2. SAVE and RESTORE macros.

The interest in this example lies in three areas

- the macros communicate with Liner; in particular they use and reset two of its variables
- the operations could not be performed by textual-substitution macros since there are no built-in Liner commands for manipulating variables
- the RESTORE macro gives a meaningful error message, expressed in terms of the Liner environment.

The last of these points requires further explanation as it uses some SUPERMAC public objects that we have not explained. These are *Merr*, which is the file to which error messages go (normally the terminal), and *Mli*,

which is a function that returns the number of input lines so far processed. (Dressing *Mli* up as a function means that this quantity is read-only.)

Note that neither SAVE nor RESTORE generates any text as output; they are therefore each replaced by the null string. They can be called by a line such as

```
/SAVE
```

which saves the environment and then executes the null command (which does nothing), or by a line such as

```
/SAVE    i  20
```

which saves the environment and then sets the increment to 20.

While discussing inter-communication between macros and Liner it is worth returning to our built-in R macro. The second line of the procedure corresponding to this macro was

Mstr ('/i 1'); *Mln*;

This can be improved by a better interaction with its environment. The problem is that, as it stands, it assumes the trigger character is '/'. If the user changes the trigger character the macro will not work any more. This can be remedied by using the *trigger* public variable, which gives the current setting of the trigger character, and the SUPERMAC procedure *Mchar*, which outputs a *char* value. As a result the above line can be rewritten as

Mchar(*trigger*); *Mstr*('i 1'); *Mln*;

With this improvement the R macro automatically uses the current trigger.

DISPENSING WITH BUILT-IN COMMANDS

As we have defined it the R macro achieves its effect of setting the line-number and increment back to 1 by outputting the pair of Liner commands

```
l  1
i  1
```

It could equally well have been made to set the line-number and increment directly, i.e.

```
procedure Rproc;
begin
  linenumber := 1;
  increment := 1
end;
```

Furthermore we can define a macro *l* that has exactly the same effect as the built-in Liner command *l*, i.e. it resets *linenumber*. (We have not actually covered the SUPERMAC mechanism for converting arguments of macros into integers, which is needed for the numerical argument of this *l* macro, but it is not hard to do.) Similarly we can define macros that achieve the effect of other Liner commands.

Taking this to an extreme we could have defined Liner to have *no built-in commands at all*. We could also dispense with built-in macros. Instead all the necessary variables and procedures could be made publicly accessible; then user-defined macros (such as our *l* macro) or macro libraries would be used to manipulate these variables.

Such a scheme can have its attractions if different sets of users want to see the same piece of software in a different syntactic or semantic light. There could, for example, be a baroque set of macros that presented the user with an array of line-numbers rather than a single one, one element of this array being active at any one time.

Nevertheless this is a course to be followed with care: the extensible language, able to be moulded to any user's needs, has always proved a mirage. One cause of this is an obvious and fundamental limitation: macros can only interact with a software component if it is executed at the same time as macro processing. If, for example, Liner had been implemented in two stages:

(1) *compilation*: the Liner commands are translated into some intermediate language
(2) *execution*: the intermediate form of the commands is then executed and the numbered output produced

then, assuming macro processing takes place at the first stage, there can be no interaction between macros and the variables active at the second stage. Thus macros cannot use variables such as *linenumber* and *increment*, which are variables that apply at execution time.

This is, of course, a well-known limitation with macros embedded in compilers. A compiler is often broken down into a sequence of stages typically there are four or five stages—and each stage may need its own macro processor (see Solntseff's[6] classification).

PERFORMANCE

If a user is defining some new macros, he needs to compile the Pascal program that defines his macros and link this with SUPERMAC and with Liner in order to generate an executable program. This is a relatively slow process—it takes about three seconds of processing time on a VAX/780 if the Pascal program is a hundred lines or so. However if the macros are to be re-used this executable program can be saved, thus avoiding the overhead of compiling and linking again (though at the expense of a lot of disc space). In this case—and it is the most frequent case since the average user does not define new macros for every run—performance depends only on the speed with which

(a) macro patterns are recognized
(b) procedures corresponding to macros are executed.

SUPERMAC is relatively slow at the former because it allows for elaborate patterns, but is quick at the latter since all the procedures are compiled, not interpreted. Certainly the overall process should be faster than a completely interpreted macro scheme.

To put speeds in perspective, it takes 15 seconds of VAX/780 time to read in the macro definitions for the **ms** macro package of **nroff**,[7] and the overhead of actually recognizing and executing the macros comes on top of this.

CONCLUSION

The prototype works. It is, we hope, not too great a leap of the imagination to assume that it would still work if

Liner were replaced by a piece of production software, be it a compiler, a text composition system or a circuit layout package.

The advantage of SUPERMAC to an implementor is that it is an off-the-shelf component which is powerful enough to meet most specifications.

The advantage of SUPERMAC to the macro writer is ease of learning: if he knows Pascal (or whatever other language variant of SUPERMAC is used) he already knows how to define macro replacements; there is thus no new syntax and semantics to learn. He does, however, need to learn how to define macro patterns within *Macrop*, etc. Even this modest amount of learning reaps further rewards if SUPERMAC is used as the macro processor in a number of different pieces of software. The only new matters to learn for each such piece of software will be the public objects (e.g. *linenumber* in Liner).

Beyond the macro writer there comes the naïve user: someone who is content to use existing macro libraries and has no desire to learn how to define macros. SUPERMAC gives this person the advantage of good performance, certainly in comparison with conventional interpreted macro packages.

REFERENCES

1. J. F. Ossanna, *NROFF/TROFF User's Manual*, Computer Science Report 54, Bell Laboratories, Murray Hill, N.J. (1976).
2. O. M. Lecarme, Review 40454, *Computing Reviews* **24** (7), 283–284 (1983).
3. P. J. Brown, SUPERMAC—a macro facility that can be added to existing compilers. *Software—Practice and Experience* **10**, 431–434 (1980).
4. P. J. Brown and J. A. Ogden, The SUPERMAC macro processor in Pascal. *Software—Practice and Experience* **13**, 295–304 (1983).
5. N. H. Gehani, An electronic form system—an experience in prototyping. *Software—Practice and Experience* **13**, 479–486 (1983).
6. N. Solntseff, Classification of extensible programming languages. *Information Processing Letters* **1** (3), 91–96 (1972).
7. M. E. Lesk, *Typing documents on the UNIX system: using the –ms macros with Troff and Nroff.* Bell Laboratories, Murray Hill, N.J. (1978).

Received July 1983