Volume 1 **C Vu** Issue 3

The Journal Of The C Users' Group (U.K.)

EDITORIAL

Good grief, issue three already! This is also the biggest issue so far, and if EVERY member of the group made the effort to write something for C Vu, we could quite easily get up to a fifty page issue in time for Christmas (1988, that is).

As usual, a big welcome to all new members, especially those who DON'T own a PC or compatible (editorial bias creeping in again!). If you have an Amiga, ST, Mac, Archimedes, or whatever why not write a review of your development software and the operating system/environment you use? If you are feeling really brave, why not try porting some of the source library volumes, and tell us what problems you encounter?

As you are probably aware, until now the C Users' Group (U.K.) has been run on an informal basis, with most of the hard work being done by Martin Houston up in Birmingham. Now that membership is into three figures it is time to go legit! We will be holding the first annual general meeting of the group on Saturday, 2nd July (provisionally) somewhere in the UK (but probably Birmingham). It really is essential for as many members as possible to make the effort to turn up, and hopefully take responsibility for one of the committee posts. Further details inside.

Phil Stubbington
Editor

STOP PRESS...

Laser C For The ST

Just arrived, although too late for this issue, is the long awaited (well, by me, anyway!) "Laser C" from Megamax Inc. for the ST. The main change between this and previous versions from Megamax is a full featured shell (with RAM cache, access to all the usual file management routines, exception trapping with backtracking.....) with integrated editor (very fast scrolling, no 32k limit on files). Also includes a whole host of utilities - all the Unix usuals (ls, egrep, make, ar, etc.) - including an extendable debugger. Libraries are _not_ up to the draft ANSI standard although the compiler itself does support enumerated and void types - hope for the near future perhaps? A full review will appear in the next issue(s)

Simple C?

Also just received is a new title from David Fulton Publishing entitled "Simple C" - aimed at converts from the BASIC lingo. Cost is £11.95 ISBN: 1-85346-057-5). Author: Ian Sinclair. Review in next issue

Surcharge Dropped

Finally, the surcharge for the 3 1/2" disk version of C Vu has been dropped!

Minix - Early Notes

By Don Forbes

MINIX - a contraction of MINIature and unIX. What are your expectations? Perhaps it will be useful to begin with a description of what you get, and then assess its immediate usefulness.

The basic system consists of the documentation (Operating Systems; Design & Implementation. By Andrew S. Tanenbaum. Publisher: Prentice-Hall International. ISBN: 0-13-637331-3) and eight disks (5 1/4" 360k). One of these is for booting, and the rest contain various parts of the system. There are several options for different styles of machine (256k, floppy only, etc.)

The system, once running, has the kernel in memory, a ram disc (\tmp) and is a complete equivalent to Unix System 3, Version 7. This version was widely regarded as one of the best stages in development. The Bourne shell is also provided.

Questions Arising

Q. What would I use it for?
A. As a (very) cheap startup and development system for Unix programming

Q. How good is it?
A. Excellent; known bugs are minimal - we are collecting these, but none to start with. Depending on your background, Minix will or will not be a culture shock!

Q. How much does it cost?
A. Less than œ100 for the book and disks. This translates as less than œ600 for complete working Unix-style system on a PC clone. The book is, of course, available seperately too.

Q. What is the legal situation?
A. To quote the code listings for Minix....

"Copyright (C) 1987 by Prentice-Hall Inc.
Permission is hereby granted to private individuals and educational institutions to modify and redistribute the binary and source parts of the system to other private individuals and educational establishments for educational and research purposes. For corporate or professional use, permission from Prentice-Hall is required. In general, such permission will be granted, subject to a few conditions"

In general, additions will probably be welcome, casual abuse will not. The source code for the compiler (constructed using the Amsterdam Compiler Kit) is not available from Prentice-Hall, but from:

Transmediair Utrecht BV
Melkweg 3
3721 RG Bilthoven
Holland
Tel.: (30) 78 18 20

Implementation Details

This section and the next are deliberately kept short, because it is still very early in the author's understanding of both the system itself and the actual implementation.

The Minix kernel is fundamentally different from a normal Unix kernel. It is much smaller and major subsystems are now implemented as such (eg. filing system). In relation to the PC (IBM PC, XT or clone) the Bios is not used at all. This is because the Bios supplied for these machines is not interrupt driven - tasks run to completion even if the rest of the system is hanging. Obviously this is not viable in a multi-tasking, multi-user environment. Hence all the device drivers control their hardware directly - screen, floppy, etc.

The system should run without too much trouble on an ordinary IBM PC with or without a hard disk. For an AT, the memory management facilities of the '286 are used (good news!) and quite reasonable multi-tasking with inter-task protection is a welcome feature. As far as is known, no special development has been made for '386 systems - the scale of this is not yet properly assessed. Please write in if this aspect is of interest to you.

Mention has been made of an implementation for the Atari ST, but details are not yet known.

Please also write in if your interest is in real-time systems, as well as Unix or others; the IEEE Posix standard is becoming a reality and I consider the development of code in relation to this the highest priority at present. There are also some major implications for the Intel RMK ('386)

Finally....

A realistic note - getting this system running on your PC may take quite a while and does require a degree of commitment. Particularly, at present, hard disk interfaces are regarded as "difficult". Do please try and report difficulties (and solutions!) as you encounter them.

Good Luck!.....

Availability

Minix is available in a number of forms:-

- a set of diskettes for the IBM PC set of diskettes for the IBM PC (or true compatible) with 640k RAM. Consists of the bootable system and the complete sources (including a number of utilities, but not the C compiler sources - details of how to obtain these are mentioned elsewhere in this article). Can be used with 512k, but program sizes will have to be adjusted.
  ISBN: 0-13-583873-8
- a set of diskettes for 256k IBM PCs. As above, but the C compiler executable is not included.
  ISBN: 0-13-583881-9
- a set of diskettes for 512k IBM PC/ATs. Supplied on 1.2M disks, otherwise identical to version for 640k PC (except RAM disk is slightly smaller).
  ISBN: 0-13-583865-7
- a nine-track, industry standard 1600 bpi magnetic tape in Unix "tar" format. Contains all the sources, an IBM PC simulator, libraries, and programs to enable the file system to work on the VAX or other Unix minis. Ports to other systems should not be too difficult, as the file system and test programs are in C.
  ISBN: 0-13-583899-1

Any correspondence regarding Minix should be addressed to Don (who has kindly offered to be the official Minix coordinator within the group):

Don Forbes
35 Upland Road
South Croydon
Surrey
CR2 6RE

Tel.: 01 688 5794

The ANSI Standard For C

A summary of the proposed ANSI extensions to the C Programming Language
By Steven W.Palmer

Introduction

Since it's introduction in 1972 by Dennis Ritchie at Bell Laboratories, C has matured over the years with each new implementation. A number of features designed to provide the language with stricter type checking and to cater for the increasing sophistication of the underlying hardware have evolved from different companies in different formats. The new proposed ANSI standard attempts to bring together the more popular and versatile extensions with a totally revised reference manual which removes many of the ambiguities of the early one. This article looks briefly at each of the new ANSI extensions.

Trigraphs

To cater for systems supporting only the ISO 646-1083 character set which provides only a sub-set of the full ASCII character set, ANSI allows the unsupported characters to be represented by trigraphs. A trigraph is two consecutive question marks, followed by a representable character.

The proposed trigraphs are shown below

Trigraph	**Character**
??(	[
??)	]
??-	~
??<	{
??>	}
??!	|
??'	^
??=	#
??/	\

The new escape code \? has been provided to prevent strings that resemble trigraphs from being expanded. For example

```
 puts("The trigraph of [ is \??(");
```

without the escape character, the ??(would be converted to [.

Reserved Words

The keywords, ASM, FORTRAN and ENTRY, are no longer reserved (although ASM is preserved in the C++ reference manual, and may later be re-introduced back into C.) The keywords CONST, VOLATILE and SIGNED have been introduced. CONST is a type modifier that indicates that the identifier has read-only properties (i.e. it may not be assigned a value after initialisation.) VOLATILE specifies that the identifier is subject to external changes, and must not be changed by the optimiser. SIGNED is already present in most compilers, and complements UNSIGNED.

Escape Codes

The standard set of character codes is extended with \a for the BELL character, and \v for the VERTICAL TAB character. Numeric escape codes now support hexadecimal codes in the format \xnn or \Xnn where nn is a two-digit hexadecimal number. Both of these extensions are already implemented in most modern C compilers.

The use of an escape character followed by a newline to indicate line continuance was originally restricted to macro definitions and strings. ANSI now allows it to apply to any line of C source code.

Strings

Strings may be concatenated by following the terminating quote of one string with the initial quote of another, and with only whitespace in between. For example

```
 puts("This is a very long string that has to be split\
 over two lines to fit");
```

can now be written

```
 puts("This is a very long string that has to"
 "be split over two lines to fit");
```

The decision as to whether similar constant strings share the same storage space has not been resolved, although it is still considered bad practice to modify string constants. Microsoft C V5.0 supports string concatenation.

Preprocessor

The following preprocessor directives have been added; #error, #pragma and #elif. #if has been extended to allow the use of the 'defined' keyword in preference to #ifdef. #error forces it's argument to be written to the user console. This allows users to implement their own error checking at compile time. For example, if a program must not be run with a STACK size of larger than 30000 bytes, then

```
 #if STACK > 30000
 #error "STACK too big - Truncated"
 #undef STACK
 #define STACK 30000
 #endif
```

will detect this and inform the user. The actual behavior of #error in relation to the compiler is mostly compiler dependent at the moment.

#pragma allows options to be passed to the compiler from within the source code. The choice of options is manufacturer dependent, and will vary between systems. As an example, Microsoft C V4.0 provides a single #pragma to toggle generation of stack checking code. #pragma check_stack+ causes the compiler to include code to check the stack on entry to subsequently declared functions to ensure that there is enough room for local variables. #pragma check_stack- switches off stack checking code.

The preprocessing rules have been rewritten to remove the ambiguities inherent in the old compilers. One notable change is that ANSI now allows the use of a macro name in it's definition. Where this occurs, the macro name will not be expanded. This allows the use of a macro to replace or redefine an existing function or macro. For example

```
 #define sqrt(x) (((x) < 0) ? 0 : sqrt(x))
```

Here, if the sqrt macro is passed a negative value, then it will implicitly return 0, otherwise it will call the actual sqrt function.

Predefined Macros

ANSI specifies that the following macros are predefined:

`__LINE__`	The current source code line number.
`__FILE__`	The source code file name.
`__DATE__`	The date at the time the macro was translated in the form "mmm dd yyyy"
`__TIME__`	The time when the macro was translated in the form "hh:mm:ss"
`__STDC__`	A predefined macro with a non-zero value

Stringization

Stringization, and token-pasting described in the next section, are already supported by Microsoft C V5.0 and later versions. ANSI allows macro arguments to be converted to strings in the expansion by prefixing the argument name in the definition by the single # character. For example

```
 #define ASSERT(n,l) puts("Error " #n " in line " #l)
 ...
 ASSERT(90, __LINE__);
```

expands to (assuming __LINE__ is 100)

```
 puts("Error " "90" " in line " "100");
```

which, after string concatenation, is equivalent to

```
 puts("Error 90 in line 100");
```

Token-Pasting

Token-pasting is the merging of two disjoint tokens in the definition to create a single new token. The inclusion of ## between two objects forces the preprocessor to combine the two objects after all macro expansion has been performed. For example

```
 #define gencode(x) callgen##x()
```

If the following is encountered in the source code

```
 gencode(12)
```

it will be preprocessed to

```
 callgen12()
```

Numeric Constants

The single type modifier letter, 'L' or 'l' which was used to cast an integer to a long integer is now complemented by 'U' and 'F'. 'U' can be used with any integral constant, and casts the type to unsigned. It may be used in conjunction with the long-integer modifier letter.

'F' can only be used with floating point values, and changes the default type of a floating point number from double to float. A floating point literal is always assumed to be of type double by the compiler.

However, the proper use of function prototypes eliminates the need to use a type modifier at all. Where needed, a cast is more explicit.

The type, LONG FLOAT, is no longer supported. An additional type, LONG DOUBLE, has been introduced. It's range is implementation defined, but should be the same as, or longer than, DOUBLE.

Generic Type

The type, VOID *, has been included as a generic type which can be cast to any other valid C type. For example

```
 char *malloc(unsigned int);
```

has now been replaced by

```
 void *malloc(unsigned int);
```

The size of VOID * is virtual, so the use of SIZEOF with VOID * has no defined meaning.

Initialisation

Unions may be initialised. The value assigned to the union must be the same type as the first member of the union. For example

```
 union {
 long Vid_IO_Ptr;
 char *VidPtr;
 } VidMap = 0xb8000000;
```

Volatile

VOLATILE is a new type specifier that is really only of interest to the optimiser. It specifies that the object it declares is volatile. In other words, it's value may be changed by an external event during program execution. For example

```
 volatile int i;
 ...
 set_vsync(&i);
 wait_vsync(); /* Synchronise with first VSYNC */
 i = 0;
 while (i) {
 /* Do odd-jobs while waiting for vertical sync */
 scan_keyb();
 check_comm();
 }
```

In the above code, taken from an actual program, the optimiser would detect that identifier i was not altered inside the loop body, and would optimise out (remove) the body of the loop! The use of VOLATILE warns it that the identifier is modified by forces outside the programs control.

Constants

The introduction of the CONST keyword is hoped to reduce the use of #define to denote constant values in a program. The use of CONST in place of #define is preferred as it conveys far more information to the compiler about the programmers intentions.

Including CONST in a declaration specifies that the identifier does not (and should not) change value during program execution. A constant must be initialised at declaration. Any attempt to subsequently change it's value will generate a warning from the compiler.

Unfortunately, without proper protection, there is nothing to stop an external library function from modifying a constant when passed the address of that constant. For example

```
 const int p = -1;
 ...
 n = libfunc(&p);
```

if the library function, libfunc, modifies the value of p, then the problem will never be detected. There is no way, in C, to indicate that a function taking a pointer argument will modify the data to which the pointer points. Unless the programmer is careful, the use of the above code will cause problems if the constant was placed in ROM or the hardware traps write-access to constant storage.

I would have liked to have seen the use of VOLATILE extended to function declarations to indicate that a function modifies external data through the use of a pointer. For example

```
 int libfunc(volatile int *);
```

Operators

The unary plus operator has now been incorporated into the C language with the same precedence as the unary minus. Other than this, the set of operators and their associated precedence have not been changed.

Run-Time Library

The standard C library has been enhanced with extra functions proposed by ANSI. Many of these functions are already available in many compilers, most notably Microsoft which has very close association with the ANSI committee.

Conclusion

In these few pages, I have attempted to outline the major extensions provided by the proposed ANSI C Standard. At the time of writing, the standard has still not appeared, and is not expected to for many more months. However, looking at the range of extensions supplied with the new Microsoft C V5.0, it would probably be safe to assume that all these extensions will appear in the final standard with little or no change.

A more complete coverage of the proposals is given in the standard C reference manual for all serious C programmers:

C: A Reference Manual. Samuel P.Harbison. Guy L.Steele.
Prentice Hall Software Series. Second Edition.

The book costs over £20, but is very good value for money. Make sure that you get the second edition.

The OS-9/68000(tm) Upgrade for the Atari ST(tm)

By Andrew Lindley

INTODUCTION

To follow up their "OS-9/68000 Upgrade for the BBC" Cumana have brought out the "OS-9/68000 V2.1 Upgrade for the Atari ST". Like Swifte's Mirage Operating System it provides multi-tasking on the Atari ST and Mega-ST without graphics (ED - GKS Level 0a is currently available for OS-9, and Level 2c is "in the works". The Mirage standard terminal driver has some limited support for line graphics). Unlike Mirage, OS-9 comes as an all in one package, however, this means a larger initial cost.

The package includes a cartridge with a boot ROM and real time clock, plus several disks (how many depends on your disk capacity) on which come:

- OS-9/68000 operating system and commands macro assembler/debugger
- 'C' compiler
- Pascal compiler
- BASIC pseudo compiler
- Stylograph (a word processing system)
- Dynacalc (spreadsheet)
- Sculptor (4GL and database system)
- MicroEMACs
- Autortc and setime (GEMDOS utilities)

The disks are not copy protected, however, all but one are in OS-9 format and OS-9 will not run without the cartridge.

With the package come three A5 ring binders containing the manuals which, at four inches thick, means you have some reading before you can use all the software. There is an optional Technical Reference Manual, which is a must for 'serious' (ie. systems) programming. I shall cover each piece of software in turn.

THE OPERATING SYSTEM

HISTORY

OS-9/68000 is a multi-tasking, multi-user operating system developed for Motorola by Microware Systems from Iowa. The original OS-9 was developed for the 6809 processor. The design was based on Unix(tm), it exploited the ability of the 6809 to run re-entrant position independent code to give multi-tasking in 64K-2M. The 64K version was ported onto the Tandy Color Computer and the Dragon where it has developed a loyal band of followers. Naturally when the 68000 was announced Microware was commisioned to implement OS-9 for it, with the greater processor power and memory further steps were made towards Unix compatibility and the current version (V2.1) which is provided by Cumana contains most of the features found in Unix.

INTERNALS

There are several concepts built into OS-9 which contribute to it being a Unix-like programming environment, they also help to explain why OS-9 is not exactly like Unix.

Memory Management

OS-9 manages memory by the use of 'memory modules'. No program or constant data can be loaded into memory unless it is in OS-9's module format. This format is, roughly, a header containing control information, the module contents (program and/or data) then a CRC which is used to spot overwrites etc.

One of the pieces of information that can be taken from the header is the module name, using this OS-9 keeps a directory of all the modules currently loaded into memory. All modules are re-entrant (they do not modify themselves) and the program's data (like GEMDOS BSS) area is not allocated contiguously.

OS-9's modular system means that it can make efficient use of the memory available when multitasking. No matter how many times you have a program running, at the same moment, only one copy of the program is loaded. It also permits two rather nice features of OS-9.

The first is the 'sticky' module; normally a module is only kept in the directory whilst it is being run by at least one process, with sticky modules the module is kept in the directory until the memory is needed. Most of the commands are 'sticky' modules and as such it is rather like having a large and fast disk cache.

The second nice feature is the 'user trap handler'; OS-9 uses the 68000 TRAP #0 instruction for its system calls, the other TRAP numbers are free for 'user trap handlers'. A user trap handler is a module which can be linked by a task to allow access to commonly used functions called by means of a trap call.

An example of this is the 'math' module on the ST; this contains the support code for all the common floating point and character conversion routines, the 'C' and Pascal compilers optionally generate code which uses this trap handler. Of course this saves memory, but more importantly it allows the math module to be changed without recompiling code, so an add-on floating point processor can be used with the change of just one module.

I/O

Like Unix OS-9 has a unified i/o system, all files and devices are treated as a stream of bytes. OS-9 divides devices into 'classes' within which they have similar function, each class is managed by a 'file manager' module (q.v.). There are five file managers available from Microware, RBF (Random Block File == disk manager), SCF (Sequential Character File == terminals, printers, modems), SBF (Sequential Block File == Tape streamers etc), Pipe (Unix like pipes), and lastly NFM (Network File Manager). The Upgrade comes with SCF, RBF and PIPE.

SCF is used for the ST console, RS232, printer and Midi ports. It provides line editing, one line retrieval, XON/XOFF etc. The ST console driver has been writtem for alpha only and is consequently fast. The keys are user defineable, the mouse can be enabled (it behaves as the cursor keys), and in monochrome a 50 line mode is available.

RBF is used for floppy, hard and RAM disks. The floppy driver allows you to define step rate and a variable sized RAM cache. The hard disk device driver uses multiple sector transfer to get speed as opposed to a cache. The RAM disk can be configured to be from 64 to 512k.

Disks have a Unix-like hierarchial directory structure (though no 'links'), filenames can be up to twenty eight (yes, 28 characters) long. File access control is similar to Unix though for compatability (with OS-9/6809) reasons only group and public permissions are implemented. Unlike Unix a sudden power down of the machine will not damage the file structure. One 'quirk' is that their is no 'root' ("/") device because each device has to have a name, so absolute paths have the form:

```
/<device>[/<dir>].../<filename>
```

For example '/h0/sys/termcap' is the absolute path to the system 'termcap' database file. This could cause difficulties with directories for '.h' files etc. However, there is a 'system default drive' ('/dd') which you can define as a duplicate to be the internal floppy, RAM or hard disk where such files are assumed to be kept.

Needless to say the OS-9 format is not compatible with the GEMDOS format, unhappily at the moment no utility is provided to access GEMDOS disks, though I am told one is due later this year. Furthermore the 'standard' OS-9 formats are either all single density or single density on track 0; as you may be aware the ST is unable to read or write in single density. However, the format used is identical with that used by the Color Computer and Dragon so many OS-9 suppliers are equipped to provide software in this format, if not (or if you're an existing OS-9 user) Cumana offer a copying service.

OS-9 PIPEs are true Unix-like pipes, both unnamed and named pipes are available, their is a slight difference from Unix in that they are by default only 90 bytes size, however, this default can be customised and can be changed to any size by the 'opener' of a pipe.

Although SBF and NFM are not supplied by Cumana, you can buy them from Microware (or their UK Agents Vivaway) and provided you are willing to resort to machine code both tape streamers and Networking are possible.

Signals and Events

These are two more ways of communicating between processes. Signals are the familiar Unix mechanism of 'interrupting' a task. The nearest equivalent to events is semaphores.

COMMANDS

Like Unix the 'shell' (command interpreter) is just another program. In fact most of the commands are not built into the shell but are seperate programs in their own right.

The familiar '$ ' prompt belies the fact that despite the 'it is very similar to UNIX in operation and appearance to the user' claim this is most certainly NOT Unix. The command set is a subset of Unix (no lexx or yacc) and the names of the common commands are not the same, for example the command to list a directory is 'dir' not 'ls', and the command to change data directory is 'chd' not 'cd'.

There is no 'batch' language, the shell will execute files as though they were a list of commands typed in at the terminal but there is no support for flow control. Some support for command line parameters is built into a utility called 'cfp' (command file processor) but it is very limited. You do get I/O redirection though you can only redirect the three standard I/O paths. They are refered to as '<' '>' '>>' (stdin, out and err respectively), append to and delete before are obtained by using '+' and '-' so '>>-listing' appends the stderr output to the file 'listing'. Wild cards are limited to '*' and '?'. Having said this, the source to some Unix like commands are available from the CUG(UK) library, and you could port a Unix shell to OS-9 with ALL its features.

PERFORMANCE

Whilst the system commands for OS-9 were written in 'C' the actual operating system itself was hand-coded in assembler. As a consequence of this and the fact that OS-9 does not page memory the performance of an OS-9 system is far better than a Unix system running on the same machine (infact Unix wouldn't run on a floppy on system as OS-9 does).

MACRO ASSEMBLER/LINKER/DEBUGGER

ASSEMBLER

This is a competent conditional macro assembler (though as is usual it doesn't support ATV's). Two special directives PSECT and VSECT are supported so that the assembler can generate files suitable for the linker to build into OS-9 module format (q.v.).

LINKER

A linker, it doesn't require you to write a control file first (as does the GST Linker, under GEMDOS). Also its documentation has the decency to tell you the format of the input file, so if you really wanted you could write your own linker for a target system. The '-g' option causes a symbol table module to be generated as well as the output module, this is used by the debugger.

USER-STATE DEBUGGER

This is called the user-state debugger because Microware also have the system-state debugger, the difference is that the user state debugger is for 'normal' programs and all other operations carry on in the background as usual. The system state debugger is for developing system code and when this debugger waits for input so does the whole system.

The user-state debugger allows debugging of a program with, providing the symbol table module is available, access to all the global variables. Not only can you set breakpoints and change memory values, you are able to 'trace' execution seeing each instruction as it executes or execute a number of instructions at a time. All this happens without the rest of the system being affected because of the operating system's support for a process running in a 'debug' state. Whilst there is no macro facility it is a very useable debugger with suitably brief commands.

C COMPILER

The OS-9 implementation of Microware 'C' is included in the upgrade, this is the current version. The C is a full K&R implementation with one restriction, bit fields within structs and unions are not supported. Microware have stated that they intend to implement this in a future release.

Some of the ANSI extensions have been added (enum and some of the functions), though void is currently #define'd to int. A full set of termcap access functions have also been added. The preprocessor directive #if xxxx is supported, so is in line assembler with #asm (though there is a strong warning in the manual). The preprocessor also supports multiline macros.

Because OS-9 code has to be position independent the amount of static data that can be referenced is limited by what can be addressed with the index register plus 16 bit displacement mode of the 68000. As a result a new 'remote' storage class has been added that allows reference to static items outside the initial 64k. In reality not many programs need more than 64k of STATIC variables and there is no restiction for automatic or 'malloc'd variables.

The code produced by the compiler seems relatively efficient, unlike the Lattice Compiler the first 2 parameters are passed in the registers. There is also an optimiser which improves the code by a claimed 11% in speed and size.

Like the K&R compiler the compiler is split into seperate programs cc (executive), cpp (preprocessor), c68 (compiler), o68 (optimiser), and then the assembler and linker are called. Make is also included and the system is a great improvement on the GEM environment for C development.

One of the claims made for OS-9 is that it is "compatible with Unix at the C source code level". Having ported Unix C-Kermit to OS-9 I can say that it is not as compatible as you might expect. The following had to be changed or added for C-Kermit.

- All stat() & ioctl() calls and the associated settings had to be changed.
- There is no fork() call in OS-9, though it can be emulated using the OS-9 equivalent of execve().
- The mode values for open etc. have different numeric values.
- There are different valid characters in filenames.
- signal() had to be emulated as there is normally only one signal vector in OS-9.
- alarm() had to be emulated, the system call is not yet implemented in OS-9.
- \n == \r (0x0d) in OS-9, so some case statements had to be fixed.
- access() fails for an existing directory unless the directory access mode bit is passed.

You might get the impression that Unix compatibility is not total, however, it has been possible to emulate all the Unix function in the OS-9 port of Kermit. The only changes to the non-system dependent code have been to change things like the #define of the end of line character.

PASCAL COMPILER

The Pascal compiler is completely ISO Level 0 compatible. This means it does not support conformant arrays (more about this later). The many extensions to the ISO standard are listed below:

- OTHERWISE selector on CASE statement
- identifiers can contain '_' and '$' (U.S. Dollar sign)
- EXTERNAL procedure directive
- 16 bit INTEGER and 32 bit LINTEGER with automatic coercion from one to the other.
- STRING data type with procedures DELETE, INSERT, STR and functions LENGTH, COPY, CONCAT, and POS.
- ARCSIN, ARCCOS, TAN and LOG functions
- Random access to files with UPDATE, REPOSITION, GET, PUT, READ, and WRITE
- Array procedures and functions SCAN, CAP, FILLCHAR, MOVELEFT and MOVERIGHT.
- Flow control for both I/O and maths error conditions.
- Can call operating system with OS9 procedure.
- Bit field manipulation (FIELDGET and FIELDPUT).
- Extended heap management.
- Access to command line arguments (like argc and argv).
- Seperate compilation
- Listing control flags

All these can be disabled with a compile time option, if you wish to produce portable code.

As I said conformant arrays (the ISO Level 1 feature) are not supported but you will have noticed that there is a new type STRING. In my experience the most popular use of conformant arrays is for string processing.

BASIC

OS-9/68000 BASIC is the 68000 descendant of the 6809 BASIC09. It is a highly structured pseudo-compiled BASIC. The BASIC command takes you into a complete BASIC environment from where you can edit, run and debug BASIC programs.

The editor is a simple line based editor almost identical to the BASIC09 editor; changes are made by either deleting the entire line or with the change (c/from/to/) command. The editor is infact the weakest point of the BASIC; though once the line is entered there is immediate partial compilation this flags any syntax errors, and the LIST command puts all keywords in upper case and auto indents loop constructs etc. for you.

Once you've typed in your program you type 'q' to quit the editor and then the second phase of compilation is carried out. This flags any context errors - if none exist you can run the program. Any runtime error will drop you into the debugger. This allows you to print off the contents of variables (or set them), single or multi step, and trace back procedures. Even better it allows you to run the trace mode (also accessible with the TRON command) this lists each source line as it is executed with the result of each sub-expression printed out beneath it.
It is infact very similar in appearance to the VM/CMS REXX trace mode.

As I said this is a structured BASIC, the structured statements available are:

```
 FOR ... TO ... [STEP ...] ..... NEXT
 WHILE xxx  DO .... ENDWHILE
 IF xxx THEN .... ELSE .... ENDIF
 REPEAT .... UNTIL
 LOOP ....
 EXITIF xxxx THEN .... ENDEXIT ....
 ENDLOOP
```

All the above are multiline constructs, and there is no necessity for line numbers, indeed you're cautioned that it slows BASIC down if you use line numbers.

User defined types are also supported with the statement TYPE eg.

```
 TYPE person := name : STRING[80] ; age, height : INTEGER;
 gender : BOOLEAN
```

and space reserved for them with the DIM statement eg.

```
 DIM fred, jim : person
```

Structure (and array) assignment is allowed eg.

```
 fred := jim (* or fred = jim *)
```

You can have REMarks for comments or use (* a comment *). It is missing the BASIC DEF FNx construct instead each "program" is a 'procedure' and the RUN statement allows one procedure to call another passing parameters in the process. The parameters are passed 'by reference' so it is in fact possible for procedures to have side effects as with C functions, however, there are no global or static variables.

Lastly, it is possible to 'pack' a procedure; this compresses the compiled form so that it can no longer be edited or debugged, but it can be run either by BASIC or by 'runb' the BASIC runtime support. BASIC has its own OS-9 module type for packed modules, so you can from the shell type a command like:

```
 $ invoice -q fred
```

and enter a BASIC program without being aware of it, as the shell will load the runtime support for BASIC and run your packed procedure for you.

STYLOGRAPH

Stylograph is perhaps best described as a 'medium' function wordprocessor. Like Dynacalc it has its roots in OS-9/6809 (versions of both are available for the Color Computer and the Dragon), however, Stylograph's roots show through a little more. Stylograph is a three and a half mode word processor. The modes are 'system', 'insert/overwrite' and 'escape'.

When you start Stylograph you are presented with 'system' mode. This is essentially a menu which you move a '>' cursor with the 'i' and ',' keys, if you press <return> against one of the options then you execute that option. The options include, call OS-9 shell, edit, print text, save text, exit, spool text to disk for later printing, read more text, load a file, save a file, and some printer setup options.

Selecting edit from system mode puts you in the escape mode. In this mode you can move around the text with 'j' for left, 'l' for right, 'i' for up ',' for down and so on. If you actually want to enter some text you have to enter either insert or overwrite modes (press ';' or '1'). This was acceptable for the old 6809 version but on a 16bit machine you expect something a bit slicker.

Once in insert mode you format the text in two ways. The first is to use keys like control-B for bold and control-U for underline. The second is using a markup language similar to ROFF to set headers and footers, at least most of the formatting is done for you to see. For example the ',ce' command causes the next line to be centred on the screen as well as on the printout. There is one noticeable ommision from the markup language - that is book type headers and footers, by this I mean when the header and footers appear like

```
 Page # Title  Chapter
```

on odd pages and

```
 Chapter  Title  Page #
```

on even pages.

With Stylograph come Mailmerge and Spellcheck. Mailmerge is a perfectly adequate system for running off Stylograph documents as standard letters. Spellcheck is a 'batch' spellchecker with a extendable dictionary. By 'batch' I mean that you have to save your document and quit Stylograph in order to run it, despite this it is adequate.

DYNACALC

I can't really comment about this as my (beta) version of the Upgrade was missing the keyboard definition file for Dynacalc (it is on its way to me at this moment). However, from the manual I can see that like its 8-bit predecessor its ancestry is in Visicalc and as a consequence it bears a more than a passing to Lotus. Though it has no macros and Label cells do not spill into the next cell. Again it can perhaps be best characterised as 'medium function'.

SCULPTOR

Sculptor is a 4th generation language and database system (once known under the name of SAGE) that is also available for MS-DOS(tm) and Unix. It would require a full article on its own in order to do it full justice. Sculptor comprises of a number of utility programs which are based around two interpreters. The first is 'sage' - this is designed to run programs which interact with the user through menu's and panels to allow database updates and enquiries etc. The second is 'sagerep' which is designed around printing out reports. There are effectively two levels of use of Sculptor; the first is using standard utilities to define a file and build a simple enquiry system and report system. The second is to write from scratch an entire application system, I'll describe the former, if you're interested in the latter then I'd go and visit your dealer.

The first job of building a sculptor database is to define the layout of its records, this is done using a program called 'describe'. This presents you with a prompt to define each fields name,title (for display), contents and size, and any validation criteria. This done, a program called 'newkf' is run to initialise the database followed by a program 'sg' which generates and compiles a sage program to allow addition, deletion, search, and amendment of the database. You then type 'sage <database name>' and you can start entering data into the database. To build a report program you simply run a program called 'rg' and this creates a sagerep program in the same fashion as the 'sg' program does for 'sage'.

MICRO EMACS

As part of the 'deal' to get OS-9 cheaply, Cumana had to forgo some things that normally come with Professional OS-9. The FSAVE and FRESTORE commands which are the 'official' method of archiving OS-9 hard disks are not included. Nor is Microware's port for microEMACS 3.6, instead Cumana have arranged to supply the OS-9 port of microEMACS 3.8i on a public domain disk with the Upgrade. This is, I am told, far superior to the 3.6 based version supplied by Microware.

MicroEMACS is a very powerful full screen editor. Commands can be 'bound' to the keys of your choice and a macro language is available. Multiple files can be edited at once and multiple views of the same or different files can be on the screen at the same time, all are edited and kept synchronised. MicroEMACS also uses the termcap file so it is possible to use it to edit in both 50 line and the 40 column modes. I am very impressed with this editor.

AUTORTC AND SETIME

These are two utilities which run under GEMDOS. The first sets the GEMDOS clocks from the real time clock (RTC) and is designed to be put in the 'auto' folder of the GEMDOS boot disk. The second is a .TTP command which allows you to set the RTC from either a command line or the GEMDOS clock, or to set the GEMDOS clock from the RTC.

SUMMARY

OS-9/68000 with all the associated software is a very powerful package. You would easily pay 50-100 pounds for an equivalent to each piece of software running under GEMDOS. However, at the new price of #497.95 inc VAT (U.K. Sterling) I expect a lot of people to be frightened off.

The obvious competition is the Mirage Operating System from Sahara, however if you were to add the price of the Mirage OS, the BASIC, Assembler, C, and Pascal from Sahara you would be paying #546.29 inc VAT. In this light the OS-9 price is good and perhaps if you want, and will use, all the rolled in software then you should consider it, certainly there is more 'function' in OS-9.

At the introductory price of #299.00 excl VAT it was unbeatable, at the current price, the question remains whether people will really believe that they want ALL the software and the limited Unix compatiblity. Given that, as with Mirage, OS-9 is best suited to a hard disk system Unix freaks should perhaps consider a PC compatible and Minix (soon to be available on the ST as well - ED) though I doubt whether it would be as reliable.

Contacts

Microware, the writers of OS-9, can be contacted at:
  Microware Systems Corporation, 1900 NW 114th Start, Des Moines, Iowa 50322
  Telephone: 515-224-1929
  Telex: 910-520-2535. FAX 515-224-1352

The Soft Centre, the UK agent, can be contacted at:
  Software House, Burr Start, Luton, Beds., LU2 0HN
  Telephone: 0582-405511
  Telex: 825115 Ref: Vivaway. FAX: 0582-456521

Cumana, who are handling the ST version are at:
  Pines Trading Estate, Broad Steet, Guildford, Surrey, GU3 3BG.
  Telephone: 0483-503121
  Telex: 859380. FAX: 0483-503326

A useful publication, if you are interested in Motorola systems in general, and OS-9 in particular, is:
  68 Micro Journal, Computer Publishing Center, 5900 Cassandra Smith Road, PO BOX 849, Hixson, TN 37343
  Telephone: 615-842-4600. Telex: 510-600-6630

REVIEW OF MIX POWER C

By Martin Houston

The first issue of C Vu carried the special offer of the MIX C compiler. This review concerns the new compiler from MIX, POWER C.

POWER C has much in common with the older MIX C. The most striking similarity is the price. The POWER C compiler costs just £24.95 (including postage & VAT). This puts it in the same 'budget' category as MIX C and Zortec (nee Zorland) C.

For the #24.95 what you get is a 670 page paper back manual cum tutorial book (a book of this size on C can easily cost #25 on its own) and two 360k 5 1/4" or 1 720k 3 1/2" disks containing the Power C compiler PC.EXE, the Power Linker PCL.EXE. The disks also contain a selection of standard header files and the C libraries in object form.

Power C uses the '.MIX' object format as used by MIX C instead of the Microsoft standard '.OBJ' format. This is no problem as the PCL linker will produce standard .EXE files at the end of it. MIX also provides an assembler that will assemble into the .MIX object format - more of that later.

So far I have concentrated on similarities between MIX C and Power C. Now some of the differences:

Firstly Power C banishes the main deficiency of MIX C - the slow speed of the code compiled with it. The original MIX C was a fast compiler but the code produced was not very efficient and therefore ran much slower than code produced by more painstaking but therefore slower systems like Zortec C.The new Power C compiler is still fast but can now produce respectably fast object code as well.

I compiled the dhrystone benchmark program (a well established test of the capabilities of a compiler or a machine to execute a 'typical' application program) under Power C and it gave a very similar dhrystones per second rating as a Microsoft C 4 compiled version of the same program. All Power C programs are optimised. The Optimiser is part of the compiler and not a separate (extra cost) program like Zortec's.

Secondly Power C is far more compatible with other C compiler than MIX C was. The compiler conforms to ANSI standards and the libraries provide many of the functions that Microsoft and Turbo C compilers offer plus numerous useful extensions for graphics, TSR writing and maths.

I tried compiling several things from the CUG source code library and experienced very few problems. The Power C compiler is strict about re-defined macros and treats one as an error. This stopped a few things compiling first time but is easily fixed by an #ifdef of #undef in the code. It will also take programs originally written for MIX C and vastly improve their execution speed. A word of warning is needed here though, as the POWER C libraries are now ANSI standard and Microsoft compatible some of the functions are different form what they used to do in MIX C. I got caught out with the settime() function which sets the MS-DOS time of day. In MIX C this function just took a string with the time in it but Power C does things the 'correct' way and uses an argument of type (struct time *).

When converting programs up from MIX C it is a good idea to cross check all the library functions used in both manuals to see if they still do the same thing.

Power C comes with sample programs to demonstrate the graphics library. The best of these is a simple program that asks for values and displays a pie chart on the screen. The graphics libraries allow programs to be written that will work on CGA, Hercules, EGA and even the new VGA screens. The Piechart program will use colours for colour screens and resort to cross-hatch patterns when the screen is mono.

The graphics support may not turn you into a video games designer overnight but it is more than adequate for business graphics of all kinds.

Another fun feature of the Compiler is the ability to write 'Terminate and stay resident' programs using functions from the library without having to resort to assembler. The manual contains an example for an on screen clock. The program compiles down to about 3k so TSRs are quite possible in C. Playing with TSRs is something that has always interested me but in the past I have been put of by having to write it all in assembler. The Power C Library does all the assembler bits like intercepting interrupts leaving the real work of a TSR in easily understandable C.

An important extra available with Power C is the source code for the libraries. This costs an amazing #10. The library source code for other compilers is usually only available for a very high price, if at all. The Power C source is complete and offers valuable extra documentation and the ability to change anything that doesn't suit you. The library sources even come with a simple assembler PCA.EXE which is used to assemble the low level assembler parts of the library. I really like the provision of an assembler. PCA is ideally suited to the short sections of assembly code needed to support a high level language. It is small and simple without any of the complex structuring required by MASM. I think that it is the only assembler that a C programmer is likely to need and is a great improvement on MIX's original policy of making you use Microsoft MASM with special macros and then converting .OBJ to .MIX with the MIX utility.

I have successfully run the Power C compiler an Apricot PC as well as IBM compatible machines so, armed with the library sources, POWER C is suitable for

Apricots and any other non-clones as most of the library function of IBM machines could be reproduced and even improved upon. The Apricot PC for example has 800x400 video and a sophisticated (by PC standards) sound chip so those areas of the Power C library could be re-written to use the features of the Apricot BIOS to good effect. The only problem with Power C for non IBM clones is that the Power C Trace debugger would not be usable.

For those of you interested in financial applications a BCD Business math library is available for a further #10. This allows calculations to take place without the rounding errors associated with the normal floating point system.

I see POWER C as an ideal compiler for the serious home user and part time software developer. The library is good enough to write serious fast executing software (with NO royalty on compiled programs). It is not in absolute terms as fast or comprehensive as a top range commercial development compiler like Microsoft C 5 but at #25 plus #10 for the library sources and assembler it offers very good value for money indeed when compared with the #400 plus needed to buy a full Microsoft C 5 system.

POWER C is available from:

ANALYTICAL ENGINES Ltd.
P.O. BOX 35,
EASTLEIGH, HAMPSHIRE
SO5 5WU

In the next issue I shall be reviewing POWER CTrace, the debugger that goes with Power C to enable the quick and painless debugging of Power C programs.

FIRST CUG(UK) A.G.M.

By Martin Houston

Up until now the CUG has been run informally by myself with the help of other members from time to time.

As membership has grown so much in the past six months the time has come to organise the group on a more formal basis with an elected committee.

I have set a provisional date for the first annual general meeting of the group as Saturday 2nd of July 1988. The venue for the meeting has yet to be arranged but will probably be in Birmingham.

The meeting will probably take up a couple of hours on Saturday afternoon.

If you wish to attend the meeting then please write to me NOW so that I can get an idea of the number of people wishing to attend for the booking of a suitable venue. I will reply to everyone wishing to attend nearer the date giving full details of the venue and programme.

The AGM will be a chance to meet your fellow group members face to face. This time no special events are planned but future AGMs could become full C programmers conferences with guest speakers and exhibitions etc.

I particularly want to hear from you if you are willing to be considered for a committee post.

Defining New Data Types In C++

By Steven W.Palmer

C++ is an object oriented superset of the C programming language, developed by Bjarne Stroustrup. The intention of the language is to make large scale program development more manageable by incorporating data abstraction, increased type checking and modularity, while retaining the low-level features of C that made it so popular. With few exceptions, any valid C program is also a valid C++ program.

In this article, I am going to demonstrate one of C++'s most widely used and powerful feature; the ability to create new data types which are totally under the programmers control. This encapulates the whole philosophy of the C++ programming language, which is to increase the ease by which a large program can be built up out of individual modules crafted by different programmers.

Standard C comes complete with a number of built-in types; CHAR, INT, FLOAT, DOUBLE and LONG. Each of these types have their own operators and the decision as to how the operators are applied is decided by the compiler when the program is compiled. For example, when adding two floating point numbers, the compiler will obviously generate different code to that used to add two integers.

C++ provides the facility to generate new types and to specify how the normal range of operators are applied to the new types. For example, we can create a new type called INTSET which is a data structure of a set of integers. This new type can then be used in the same way as the built-in types. You can use the + operator to create the intersection of two sets, the * operator to create the union of two sets and the - operator to create the disjoint of two sets.

Because the + operator is already used to add two numbers, we will have to OVERLOAD it to indicate that it will be used for another purpose. This is another aspect of C++; the facility to redefine the behaviour of existing operators. The compiler differentiates between different uses of the operator by looking at the type of the operands.

Finally, we need to define how the new types are created and destroyed. When a type is declared locally in a block, it is automatically created on entry to the block, and destroyed on exit. With the built-in types, this is performed by the compiler. C++ allows us to specify our own means, although most users stick to the conventional approach of allocating space for the type on entry and destroying the space on exit.

It would probably be easier if I illustrated this. I am going to create the new type, INTSET, which is simply a set of integers. The C++ code to do this is as follows

```
 // Define class INTSET
 class intset {
 int set[];
 int size;
 public:
 intset(int);
 ~intset(int);
 int& operator[](int);
 void operator=(intset&);
 int& operator+(intset, intset);
 int& operator-(intset, intset);
 };
```

Note: The // symbol denotes that the rest of the line is a comment.

The class enscapulates all the data and functions required to manipulate type INTSET. All definitions before the PUBLIC keyword are private to the class and cannot be accessed by the user without explicitly qualifying the class name. In normal use, the user never needs access to private data and functions.

The public class contains, in order, the names of functions to create the new type, destroy it, access one member of the set (using the standard C subscription format which has been overloaded here), assign an element to a set, compute the intersection and to compute the disjoint of two sets. As we develop the type, we can easily add extra features. For example, we can allow all the elements of one set to be copied to another by overloading the = operator again.

```
 void operator=(intset);
```

The compiler automatically selects the appropriate method of applying the operator.

The new type is declared as follows

```
 intset myset;
```

When a type is declared, the following function is called to create the new type in the current scope of the program.

```
 intset::intset(int sz)
 {
 size = sz;
 set = new int[sz];
 pos = 0;
 }
```

The other function, ~INTSET, is a destructor. It is automatically called when the program exits from the scope in which the type was created. It's purpose is to destroy the allocated type.

```
 intset::~intset()
 {
 delete set;
 }
```

NEW and DELETE are similiar to the familiar C functions, MALLOC and FREE. The argument to NEW is a single type which is used to calculate the amount of space required. DELETE returns the allocated space back to the operating system freestore. The equivalents in C are

```
 set = malloc(sizeof int * sz);
 free(set);
```

Access to the elements of the set is achieved through the following function, which is called whenever an element of an INTSET is accessed.

```
 int& intset::operator[](int s)
 {
 if (s <0 || s >= size)
 abort("Set bounds out of range");
 return set[s];
 }
```

Accessing an element is done in the same way as access to an array; by using the subscription operator. However, unlike an array, an INTSET is one dimensional.

Notice how we have added array bounds checking to our new type. In other languages, Pascal, Ada or BASIC, the processor will output an error and stop if we attempt to access outside the declared range of an array. C did not provide this, and consequently lead to some hard to find bugs. Often programs would write data all the way through memory until stopped by a reset or the memory management unit.

Assigning elements to a set is best done like this

```
 intset myset;
 myset = 12;
```

The number 12 then becomes a member of intset. There is no point in using the subscription operator since the actual position of an element in a set if of no interest. The code to assign elements to a set is as follows

```
 void intset::operator=(intset& element)
 {
 if (pos == size)
 abort("Set overflow");
 set[pos++] = element;
 }
```

Next, we overload the + operator so that it will compute the intersection of two set operands. The result of the computation is held in allocated memory, since we cannot be sure what is going to be done with it. The user may either assign or compare it with another set.

```
 intset operator+(intset x, intset y)
 {
 int xi = x.pos;
 int yi = y.pos;
 int nn = 0;

 // Allocate enough space to hold the largest set
 intset result[max(xi,yi)];

 for (int jj=0; jj<xi; ++jj)
 for (int kk=0; kk<yi; ++kk)
 if (x[jj] == y[kk])
 result[nn++] = x[jj];
 return result;
 }
```

This is not exactly the best way to do the job, but it works and can be easily understood.

Finally, we overload the = operator again so that it can be used to copy the elements of one set to another. Note the differences between the parameters of this function and the previous operator= one.

```
 void intset::operator=(intset z)
 {
 int u = z.pos;
 if (u > size)
 abort("This can't be happening!!");
 for (int jj=0; jj<u; ++jj)
 set[jj] = z[jj];
 }
```

The error message is given, because it is not possible for the intersection of two sets to be larger than the size of the largest set! It is simply included as a catchall if there was a bug in the function to handle the + operator.

Now we will put this all together in a program that uses the new type to create two sets, and build a third that consists of the intersection of the first two.

```
 set_example()
 {
 intset a[4],b[4],c[4];

 a = 12; a = 45; a = 90; a = 16;
 b = 17; b = 45; b = 16; b = 78;
 c = a + b;
 }
```

When the program enters function SET_EXAMPLE, the constructor is called three times to allocate space for three new types. Then we enter four elements into the sets using the overloaded assignment operator. The following line then computes the intersection of the two sets and stores the result in a third. Since the result of the intersection was a set rather than an integer, the compiler used the second assignment function instead of the first.

This concludes my introduction to the C++ programming language. Unfortunately, as of the time of writing, there are few C++ translators on the market. Zorland is rumoured to be bringing out a budget C++ compiler, as opposed to a translator. Such a compiler would translate the C++ source code directly into machine code, rather than into C source code. The resulting machine code will be far more efficient and compact. Hopefully, when this happens, we will see C++ begin to achieve the even greater recognition that it is deserved.

File Recovery Under Xenix

By Martin Houston

"IMPOSSIBLE!" I hear you cry. In most cases I would agree with you.

If the system is working hard then the filesystem blocks that you discard will end up in someone elses file faster than you can act to stop it.

If however you are the only active user on a machine and you 'rm' a precious program or document that you have been working on for weeks try this program before fully enjoying a nervous breakdown!.

1. If you are not already root get super user status - quickly the longer this part takes to more likely you are to loose the data.
2. Once root type 'sync;/etc/haltsys' to shut the machine down fast. Make sure you have the agreement of others logged on for this.
3. Your next action depends on which filesystem the deleted file was on. If it was on /dev/usr or another mounted filesystem then simply take the machine single user.

   In this case recover must be living in /bin or /etc so you can execute it without mounting usr. If the file was on root then some foresight is needed - you have to prepare a system boot floppy with the recover program on it. The important thing is that the filesystem with the lost file must not be re-mounted until the recovery has taken place. If ANY files are written to the filesystem blocks from the freelist will be taken to write them into. It is these very blocks that recover is interested in.
4. Cd to a directory in which it will be safe to create temporary files. If saving a big file off root from a boot floppy then mount /dev/usr from the floppy & use /usr/tmp as the work area.
5. Invoke recover with the name of the filesystem device (/dev/root or /dev/usr) and a number of K byte blocks to recover from the free list.

This number should be the size of the file that you lost plus a few more for luck.

6. You will find that recover will create a number of files in the current directory in the form flist[0-<number you wanted>]. It is now up to you to piece together your original file from these fragments. The only help I can give you for this stage is the observation that flist0 is usualy the first block of the file, flist1 the second & so on. If any files have been written since your file was deleted then the first few blocks will be missing. This is because the free list is actualy used as a stack. The most recentley freed block is the first to be allocated to a new file.

   The file is compiled with the following options.

```
cc -M2es -DM_KERNAL -o recover recover.c
```

   Happy Hunting!

ATARI ST LATTICE C V3.04.01 BUGS

(List courtesy of Metacomco plc)

The following list documents all the bugs found to date in Lattice C version 3.04.01. If you wish to obtain a disk of updates, please write to Metacomco at:-

Lattice C Updates,
Metacomco PLC,
26 Portland Square,
Bristol,
BS2 8RZ.

Please enclose one double sided disk or two single sided disks if you require all the latest updates, or one single sided disk if you only require the updated libraries.

(In the following list, a tick or cross indicates whether the bug has been attended to or not, a normal bullet - • - usually indicates a partial fix, detailed further in the text)

✓ SRAND.. and DRAND do not work

✓ CXM33 and some FP primitives corrupt registers. This is a serious bug as it has a knock-on effect on other functions.

✓ LINEA.H contains 4 bugs.

• FORK and EXEC contain bugs. As a workround, use PEXEC instead as this is called by both FORK and EXEC. PARTIALLY

✗ PUTENV often returns wrong results.

✗ STCI_H not included in CLIB as stated in the User Manual. However STCH_L is equivalent to STCH_I, STCI_H is equivalent to STCL_H and STCI_O is equivalent to STCL_O. STCO_I and STCO_L are equivalents, and neither exist in the Lattice library.

✗ LINEAC uses A6. Should use A2.

• OBJC_EDIT(tree,object,char,&index,type) should be
OBJC_EDIT(tree,object,char,index,type,&newindex)

This function was documented correctly, but was not changed on the disk.

• FCLOSEALL should be used as FCLOSEAL.

✗ There appears to be a bug in the implementation of the STDERR stream. Output to this stream will not produce the required effect.

✓ It is not possible to compile to or from the RAM disk using LC.TTP. This works fine when using LC1 and LC2 separately.

✗ Problem with STRNCPY()

✗ ACCESS returns 0 correctly when mode is set to 0, but always returns -1 when mode is set to 2, 4 or 6.

✗ STRNICMP does not seem to work.

✗ CSCANF does not operate correctly for input of floating point variables.

✗ PRINTF %g, %w, %le formats not working correctly.

✗ The escape character \f is not recognised correctly.

✗ FDATIME. Problem with parameter passing and the size of INT.

✗ Calling a function containing >32K of code will compile OK but gives 4 bombs when executing.

✗ Problem with pointers to SHORT INT

✗ * and ^ Operator priorities wrong. ^ should be higher than *.

✗ Problems with POSERR.

✓ Calls with explicit references to STDOUT treat it as buffered and untranslated.

✗ Writing and reading a long constant causes an error.

✗ VS_CLIP not working correctly.

Other bugs have been found with the other utilities sold with the compiler. These are as follows.

LINKER

• The error at the end of linking that causes the bombs usually occurs when using the -DEBUG and -NOLIST options together. Sometimes this will happen when using only the -DEBUG option. This is because in the final stages, after the actual linking has been completed, information is being written back to the wrong ddress. It is not a serious error as the resultant .PRG file is executable. In most cases, you can continue without having to reset.

• Pass 2 of the linker terminates and returns to the desktop too quickly to read error messages when using it from the desktop.

• Does not recognise PATHS in it's control file.

• Will not operate with Mark Williams or Beckemeyer shells. This is necessary because many users have these shells.

KRSC V1.1

- Not compatible with MEGA TOS (V1.09). Lattice now distribute V1.2.

ED.PRG

- Block handling causes crashes.
- Does not check if a disk is full. It simply writes as much as possible to the disk and then exits, resulting in incomplete files being sacved to disk. If there is very little or no room at all when it tries to save to disk, all of the file can be lost.
- Scrolling too fast using the UP and DOWN arrow keys can cause a crash.

DEBUG+

- Bug that causes the error message "BUS ERROR CXINIT". A patch is provided. Now fixed in V1.04.
- Not enough workspace for some applications. Use % option as a workaround. E.g. %150000 to grab 150K. Fixed in V1.04
- Does not allow LINE-A opcodes to be executed in trace mode, they are reported as illegal instructions.
- Gets confused between label/function names and it's own command words.
- Wrongly disassembled MOVE.L #$64,D1 as MOVEQ.L #$64,D1.

MAKE.PRG

- Does not work with write protection on.

MIRAGE UPDATE

By Phil Stubbington

Following on from my review of the Atari ST version of Mirage in the last issue, I have received a copy of "SoftWord" - the house journal of Sahara Software. As well as news of a new revision of the Mirage system software, it contains several other items which may be of interest to members contemplating switching to Mirage.....

Change Of Address!

As of 24th March 1988, Sahara have moved from Vauxhall to:

South Bank Technopark, 90 London Road, LONDON, SE1 6LN
Tel.: 01-922-3350

The more astute amongst you will notice that this puts Sahara on the same site as their parent company MicroAPL - who have announced APL.68000 under Mirage. The Omnis3 database and application generator is now available, as is the OxSys business suite.

Additions & Changes

The major addition to the latest revision of Mirage is an archiver, allowing the easy backup of hard disks to floppies. Some utilities (notably, the spooler) have been enchanced and bugs fixed. The Pascal compiler has also been updated.

And Finally.....

Sahara Software run a number of Mirage oriented courses for everyone from new users to system programmers. Further details can be obtained from David Eastwood at Sahara.

STRUCTURE, PART 3

By Colin Masterson

In this series we have been considering how to achieve structure in our programs and have identified some key issues:

To produce a structured program we must:-

- Concentrate initially on high level tasks.
- Recognise special cases.
- Produce simple, single entry/exit processes.

Understand the problem.

Now, we have said nothing so far about how you are solving the problem, the algorithm or methods you are using, but concentrated on how you go about laying out the chosen method.

So perhaps it is worth stating that, before you begin any sort of work, you must fully understand WHAT you are trying to do and HOW you are going to do it. The best structure in the world will do nothing if the algorithm is wrong. However, if the algorithm is wrong, if you've structured your program well, finding and fixing the problem will be a lot less painful.

Thinking back to our original problem. If we realise late on that our algorithm for opening the lid of the kettle is wrong, then we only have to change the detail INSIDE that procedure. Where it's called from, the body of the program, remains the same. Structure has helped us. We don't need to go through 200 lines of code looking for affected lines.

This early stage of thinking of the problem and deciding on the algorithm and then laying out the broad steps is crucial. It's the most difficult part of the whole process. Get things wrong at this point and you make work for yourself later on. I'm going to say that again because it's important. Get things wrong at the initial algorithm and main body stage and you make work for yourself later on. So take the time to think it through.

Here are some tips to make this easier.

- Just write out steps in plain English first.
- Play 'what if' games with the data/input.
- Think about what it will look like or actually do in the end. Is what you're planning now going to give this ?
- Allow your mind to wander off down a couple of levels at each step without getting too detailed, just to see if things seem to flow OK.
- Think about how you are handling the data. What data structures or types are you using ? How will they be passed from function to function ?
- What function needs to know what - is that data going to be available at that point in the process.

For trivial tasks these steps are easy. The more complicated the task, the more essential it becomes to do each correctly.

Having done this on the top level then really the same procedure can be repeated for lower and lower levels.

The story so far..

Structuring then makes use of several key points:-

- The program will be constructed on a hierarchical basis with each level restricted to handling tasks and data applicable to that level. The overall purpose of a level will not be obscured by detail relating to lower levels.
- Data structures and variable names will be chosen to suit the particular level.
- At the highest level only broad outline steps will be identifiable.
- Each procedure will follow a single entry, single exit pattern. Loop constructs such as for() while() and do() will be used to provide repeated sets of instructions.

  Where nesting of loops, or the number of instructions within a loop, is more than 3 or 4, then consideration will be given to creating a further sub level function to hold these instructions.

  In this way the broad purpose of the loop is not obscured by heavily detailed loop bodies.
- Functions will be laid out on the page in such a way that the use of comments, headings, indentation and whitespace make the level and purpose of the function clear.
- Procedures will be constructed as far as possible in a general purpose manner such that changes to the boundary conditions, number of iterations or data does not substantially affect program layout.

Next time we'll consider how the choice of data type can help us in structuring.

THE SOURCE LIBRARY

The Source Library is intended to be a collection of public domain C source code brought together so that any member of the C Users' Group (U.K.) who wishes to, can use and develop them as an aid to their own learning and enjoyment and to improve the stock of public domain C source.

1	Software Tools #1 - bracket and comment checkers, structure analysers, three variations upon the XREF theme, several hex loaders and dumpers, time command execution.......	8	Shells - two Unix style (single-tasking only though) shells for MS-DOS - source code included so conversion to the ST & beyond is a possibility - why not give it a try!
2	Games #1 - principally the AdvSys adventure writing system with all sources, documentation and a sample adventure. Also contains the Towers Of Hanoi, and Conways "Life"	9	C Language Tutorial #1 - this disk contains the text of the tutorial. Most of the example programs can be found on volume 10

3	Editors - one of the latest version of the MicroEmacs editor; with sources and extensive documentation. Now contains a macro language and several other enchancements	**10**	C Language Tutorial #2 - the example programs to accompany the text found on volume 9
4	Languages #1 - language compilers and interpreters in C. This volume contains the sources for XLISP; the object oriented version of the list processing language	**11**	Communications - several comms protocols (Kermit, SEAlink) and library managers (Arc, Lump, Squeeze) and some related utilities
5	Math - a comprehensive collection of the usual functions. Also contains a set of MASM macros for the 8087 co-processor, programs for numerical integration and matrix manipulation	**12**	PC Utilities #1 - some of which are provided as executables only. Contains an extensive _shareware_ window manager, cursor control progs from Bill Sparrow, EGA graphics routines,......
6	Unix Utilities #1 - pattern matching language, "make" project management tool, stream editor (complete with example sript - convert Pascal to C!) and several minor utilities	**13**	Languages #2 - two subset C compilers (cpcn and ratc) with sources, an interpreter (sci) as MS-DOS executable only, and a number of Unix style utilities.....
7	Unix Utilities #2 - text processing and printing; a couple of text formatters, entabbers and detabbers, menu driven setup programs for IBM and Epson printers....	**14**	Games #1 - the Dungeons and Dragons style game "Larn". Should be possible to get it running on most systems. Requires lots of memory to run!
15	Unix Utilities #3 - source for LEX (lexical analyser) for MS-DOS, but should be possible to convert to other OS's	**NEW 19**	Minix #1 - conversion of the Kermit comms program. Conversion and documentation (of use to any Minix devotee) by Adrian Godwin
16	Unix Utilities #4 - together with volume 17, these contain lots of material related to the LEX & YACC (yet another compiler-compiler) language systems, including......	**NEW 20**	Unix Utilities #6 - mainly a healthy bunch of benchmarks, but also a number of general purpose utilities

17	Unix Utilities #5 - some applications with parts written in lex and yacc	NEW 21	Games #2 - the one that started it all off - the Colossal Cave adventure with full source code
18	Software Tools #2 - some more programmers tools (see volume one) and a simple Unix Curses type screen library	NEW 22	WHY NOT CONTRIBUTE some of your own work, or conversions of existing library volumes to other systems? Non-IBM specific contributions especially welcome!

DISCLAIMER

The C Users' Group (U.K.) makes no representations or warranties with respect to the contents of "The Source Library" listing and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose of the library volumes mentioned herein.

ORDERING

To order any of the volumes in "The Source Library" simply write down a list of the volumes you require indicating:-

disk type (3 1/2" or 5 1/4")
your disks or ours (don't forget to actually send them!)

and send it with correct payment (a cheque or postal order made payable to C Users' Group (U.K.)) to:

The Source Library, C Users' Group (U.K.), 36 Whetstone Close,
Farquhar Road, Edgbaston, Birmingham, B15 2QN

CHARGES

The fully inclusive charges for each volume are:

£2 (your own 5 1/2" or 3 1/4" disk), #3 (our 5 1/4" disk) or £3.50 (our 3 1/2" disk)

A CALL TO QL OWNERS

By Martin Houston

I notice from group records that there is a significant number of you out there that use the Sinclair QL either as an only or a secondary machine. I have had more than one enquiry from QL owners about how they can access the CUG library as the QL cannot directly understand the MS-DOS format disks that the library volumes are supplied on.

If you are a QL owner that can access the CUG library by whatever devious means you can devise why not help out your less fortunate fellows. Why not become a SUB LIBRARIAN and offer selected library material to other QL owners on microdrive or native QL disk format?

If you would like to take on this responsibility then drop a line to Martin or

Phil and the library sub-branch will be publicised in the next issue of the newsletter.

The same idea goes for anybody else who can offer access to the library by different means than the central library source can provide. Is there still any call for all those exotic old CP/M formats? How about an Apple Librarian or a Sirius Librarian to offer the service of the library to people with those formats.?

If you want to be a sub-librarian all that I require is that you get the volumes from the library in the usual way and pass any improvements back up to the central library so that they can be shared by all. Any money that you charge in library fees is yours to keep as an incentive for running the service. Don't expect to get rich by it though!

WRITING FOR C Vu

By Phil Stubbington

As you can see by flicking through any issue of C Vu, the group relies heavily on a small group of members who make the effort to contribute (which is, after all, what a user group is all about!). If every member made the effort to write _something_ for C Vu then we would have enough material for many issues to come.

Although articles are always welcome, within the group we have many members who can help out with your technical queries - either about C itself, or related to the operating system/environment you are using. At present, we have members who can help out with technical queries on MS-DOS, OS-9, Unix, Xenix, Mirage, TOS (including GEM), Minix....

Submissions for C Vu should be in the form of ASCII text files, on 3 1/2" or 5 1/4" standard MS/PC/GEMDOS format disks. 5 1/4" BBC Micro format (standard DFS or ADFS) single-sided/single-density (ie. 100k) disks can also be accepted. The alternative is to send articles by modem, to one of the bulletin boards or conferencing systems mentioned elsewhere in this issue.

Ideally, the only formatting in text files should be a double CR/LF between paragraphs and between each line of code (if using examples, etc.) Generally speaking, source code of great length will not be reproduced in C Vu, but will be placed in the "Source Library".

Any graphics should be printed out (maximum size is 6 3/4" by 9 1/4") although standard GEM format files (IMG or GEM types) can also be accepted.

NOTES FROM THE ULTRA-CONSERVATIVE RIGHT WING

By Marc J. Rochkind

Here's a list of rules that every C programmer should follow. Occasionally, a rule may be violated, but only for a really good reason.

1. Most standard functions have a header associated with them, which should always be included. The manuals often fail to identify what this header is, although it's there in "/usr/include" (or wherever). You might consider grepping the header files to discover what headers go with what functions. This rule means that most C files will start with 5 to 10 includes. Don't feel weird if some of your files start with even more.

2. Never declare a function if it's declared in a header, as redundant declarations lessen reliability, portability, and maintainability. If you discover that some standard functions aren't in any header, you might consider

coding a header so you can just include it all the time.

3. Always declare the return value of a function when the function is defined, even if it's "int". This makes your intent clear. It also avoids letting a function default to "int" when it should have been declared "void".

4. For functions internal to a file (and, therefore, not declared in a header), order them so that redundant declarations aren't necessary. That is, order them as a Pascal program would be, with "main" at the end. Only in the case of mutual recursion is this impossible.

5. Always call "exit(0)" at the end of "main".

6. Make all variables and functions "static", unless they are external to the file. This helps the reader by indicating the locality of the object, and it also avoids name clashes in the linker.

7. Don't use "int", "0", and "1" for Boolean operations, as these are needlessly general. Instead, use these:

```
 typedef int BOOLEAN;
 #define FALSE 0
 #define TRUE 1
```

8. Where a logical expression is required ("if", "while", etc.), do not write an integer expression. That is, do this:

```
 if (*p != '\0')
```

not this:

```
 if (*p)
```

This is done to speed up debugging. When the second technique is used, mistakes are often made because the negation of the correct expression is accidentally written. This bug is perhaps most frequent with non-intuitive return values, such as the one from "strcmp".

9. Never call a function without checking its error return. If the error is impossible, at least call "assert", as in this example:

```
 if (close(fd) == -1)
 assert(FALSE);
```

10. Check null pointers against NULL, never against 0 or anything else. Do not cast NULL:

```
 if (flurb(x, y) == (struct Rcd *)NULL) /* bad */
 ...
```

11. When you have an initialization, a test, and an incrementation, use a "for" loop instead of a "while" loop. This makes the loop control more explicit and reduces the possibility of forgetting to increment.

12. When calling a function that returns a useful value or an error return, use this "assign and check" paradigm:

```
 if ((p = fcn(x, y)) == NULL)
 ... handle error ...
```

While ordinarily I would recommend against doing assignment in the middle of an expression, this is an exception. It occurs so often that it's immediately recognizable to the experienced reader, requiring no time to figure out. Non-experienced readers should learn about it.

13. Never read anything from a human user without checking it thoroughly for errors and reporting problems back precisely. For example, if the user hits Control-D or Control-Z, don't give a "conversion error" message. This rule prohibits the use of "scanf" in most cases.

14. Don't use lousy functions just because they're in the library. What some library contributors know about programming you could fit in your left ear. This rules out functions like "atol", which reports no error.

15. Do not put in gratuitous comments, because it cheapens the value of all comments. Then the reader gets into the habit of skipping comments, and, when you have something important to communicate, there's no way to do it. Also, comments make the code harder to scan, so they have to be really useful for the reader to want to pay the price.

Here's an example. First, a gratuitous comment:

```
 /* get input from tape drive */
 if (readtape() == ERROR){_
 ...
```

Now, a useful comment:

```
/* can't use gettape() -- hangs system when buffered */
 if (readtape() == ERROR)
 ...
```

16. Except for the numbers -1, 0, and, maybe, 1, always use a symbolic constant. Do not dimension arrays with numbers like 81, 257, etc. For string lengths, adopt a convention and stick with it. I use a symbolic constant to define the length exclusive of the NUL byte, and then add one for dimensioning. The alternative is, of course, also OK.

17. Every "switch" statement should have a "default" case, to reduce debugging. If it can't ever occur, use "assert":

```
 switch (x) {
 ...
 default:
 assert(FALSE);
 }
```

18. If your compiler supports function prototypes, always use them, both for library functions and for your own. They are the greatest contribution to C programming since the curly brace. I've found this paradigm useful for static functions, as it makes modifications easier:

```
 static BOOLEAN fcn(int, float);
 static BOOLEAN fcn(count, val)
 int count;
 float val;
 {
 ...
 }
```

Note that come compilers are set up to ignore library prototypes by default. For example, with Microsoft C you have to define the symbol "LINT_ARGS".

19. Find out what compiler options generate the most warnings and the strictest error checking, and turn them on. Make sure all of your files compile with no warnings at all, or else you'll miss a new warning because it's mixed in with the old ones. Doing this will often mean that you'll use lots of casts. But don't just blindly put in a cast to make a warning go away, or you might overlook an actual bug.

20. How you lay out your programs is a matter of personal preference. But at least be consistent, or you risk telling the reader that you don't know what you're doing. Either put a statement on the same line as the "if", or on the next line, but don't do it sometimes one way and sometimes the other. Put blanks between a function name and the left parenthesis, or don't, but always do it the same way. Since all bugs are introduced during text editing, that's when you need a consistent style. So, don't use a C beautifier, unless it's to fix up a program someone else gave you.

21. Call "assert" as often as you have the energy to. Don't define "NDEBUG" to remove assertions when your program is ready for delivery, because that's when undiscovered bugs have their worst consequences. Remove assertions only when the profiler tells you that you must.

Marc J. Rochkind February 7, 1987 CompuServe 75765,1233

THE SPECIAL OFFER.... FREE LIBRARY VOLUMES!

Every issue of C Vu so far has carried a special offer of some sort - and this is no exception. We are offering you the chance to receive volumes from the Source Library (detailed elsewhere in this issue) at no cost (well, *almost* no cost!)

All you have to do is submit an article, review or letter for publication in C Vu. The only rules that apply, in addition to the guidelines set out in "Writing For C Vu" are:-

- each submission must be on a seperate disk - either 5 1/4" or 3 1/2"
- the minimum length is 500 words

If your article appears in a subsequent issue of C Vu, you will receive the library volume(s) of your choice. Each accepted submission entitles you to one volume - so if you write a five part series, you can receive five volumes!

You could, in fact, get all the volumes in the library for next to nothing by writing a review of each volume (perhaps with details of any difficulties you encountered if you are porting to a different compiler, OS, etc.)

NOTE

This offer also extends to members you have already had articles published in C Vu. Just write in, enclosing a blank disk, telling us which issue of C Vu your article appeared in, an which volume(s) you would like to receive.

CUG BULLETIN BOARDS

The group has some presence on all of the following bulletin boards/conferencing systems. If you have a modem, why not give them a call and make a contribution to our conference or special interest group?

- Chronosoft Lair - 021 744 5561; 300N81; V21/23 at present
- CIX - 01 399 5252; All speeds; N81; Have your Barclaycard handy!
- Digital Matrix - 021 705 5187; 1200N81; V21/22
- Dr. Solomon's FIDO - 02403 4946; All speeds; N81

C Wizard's Programming Reference

By W. David Schwaderer
John Wiley & Sons, Inc.
ISBN: 0-471-82641-3
$19.95 (US Import)

Review By Phil Stubbington

I am always a little wary of any computer book which seems to be trying to boost the ego of the potential purchaser; I'm sure you know the sort of thing - "The Advanced Programmers' Guide To....". The Wizard's Reference fortunately doesn't fit into this category, being a highly readable and usable C reference.

In his dedication, Mr. Schwaderer mentions that this book was "only supposed to be a reference card". The book now runs to over 200 pages, spiral-bound (as ALL reference books should be!), with sections devoted to a C language overview, C operators, preprocessor and statement reference, standard and non-standard library file I/O and a number of appendices.

The book has been designed as a quick-reference guide, with white-on-black frames used to make particular areas stand out as you flip through. Each area,then has a discussion of the subject, use, examples, format, comments, and reference, followed (often) by a list of the hazards. For example, the discussion of the "char" type, explains why you should declare variables which may contain EOF as an "int".

From the blurb on the back of the back, it is obvious that one of the aims is to help create portable programs. Throughout the book, references are made to the draft ANSI X3J11 standard and potential portability problems (such as the EOF problem mentioned above) are pointed out.

Mr. Schwaderer has a very readable writing style, and extensive use of diagrams (to show the format of structure declarations, for example) actually adds something to the book, rather than just being space fillers. As a pleasant change, the book also has a good index, so if you can't spot what you are looking for from just flicking through, the index usually helps.

The Wizard's guide is well worth a look by any reasonably proficient C programmer, and is probably a useful backup to a tutorial guide for newcomers to the language. Incidentally, a similar style has been adopted for other books from Wiley - including a reference for Prolog and Modula-2 (I think)

Debugging With The Macro Processor

By Martin Houston

The C macro processor is the first pass of the compiler that converts #define symbols into the data in their definitions. An example of this is:

```
 #define TRUE 1
 #define FALSE 0

 main()
 {
       if(TRUE)
             printf("TRUE");
       else
       if(FALSE)
             printf("FALSE");
 }
```

This code is just a bit of nonsense to show how the macro processor works. Here is how it would look after the macro expansion pass of the C compiler:

```
 main()
 {
       if(1)
             printf("TRUE");
       else
       if(0)
             printf("FALSE");
 }
```

Notice that TRUE has been replaced by 1 and FALSE has been replaced by 0 (a zero value is 'false' to C). The action of this program is to print the word TRUE. Notice that the strings "TRUE" and "FALSE" have been preserved. Macro processing is not just a textual substitution such as would be done with an editor.

An important feature of the macro processor is the ability to test the value of a defined sybmol and alter how code is compiled accordingly. The most common use of this is to selectively include extra code for debugging purposes.

```
 #define DEBUG
 .......
 #ifdef DEBUG
       printf("some debugging message....");
 #endif
```

If you wish to remove the debugging printf from the program then the only thing that needs to be removed is the "#define DEBUG" as if the symbol "DEBUG" is not defined the first pass of the compiler will omit everything between an #ifdef DEBUG and its matching endif. For only one message this is little different from the work involved in placing a comment arount the printf statement to get the same result but if the program has many debugging messages instead of one then it is much quicker to control them with the macro processor.

With some compilers it is possible to supply definitions to the macro processor on the command line so the #define DEBUG need not be present in the file itself.

Unlike comments ifdef/endif pairs can be nested, each endif will match the most recent ifdef. If there are many ifdef/endif pairs in the code it is a good idea to 'tag' each endif with the ifdef it belongs to to save confusion - like this:

```
 #ifdef DEBUG
       printf("some debugging message....");
 #endif /* DEBUG */
```

The comments around the second DEBUG may not be need for all compilers but

Microsoft C for one complains about extra characters after an endif directive. As comments are the first things to be stripped out by the macro processor a comment can be put anywhere without changing the sense of anything.

What I have just presented above is the 'traditional' way to include debugging code. If more use is made of macro processor features debugging can be made easier and neater.

The most powerful feature of the macro processor is that macros are allowed to have paramerets. The best way to explain this is with an example:

```
      #define MAC(A, B, C) (A = B * C)

      main()
      {
            int v1, v2, v3;
            v2 = v3 = 10;
            MAC(v1, v2, v3);
      }
```

will exand to:

```
      main()
      {
            int v1, v2, v3;
            v2 = v3 = 10;
            v1 = v2 * v3;
      }
```

Although the use of macros with parameters looks like function calls the effect is to 'expand' into the code of the macro definition. The MAC macro above simply wouldn't work as a function. If v1 was a parameter it could not be assigned the value of v2 * v3.

The last tool we need to be aware of before building our debug tools is the 'special' macro names. There are two symbols that the macro processor defines for itself but are accesable inside other macro definitions. The macro __LINE__ (note the double underscores on each side of the definition) is the current line number that the macro processor has got to in the source file and __FILE__ is a string containing the name of the current file being processed. These are two things that are very useful to know in a debug message!

Here then is the first of my debug macros:

```
#ifdef DEBUG
#define DEBUG0(S) {printf("%s(%d) %s", __FILE__, __LINE__, S);}
#else
#define DEBUG0(S)
#endif
```

Note that if DEBUG is not defined the DEBUG0() macro has no body. This means that any DEBUG0 in the code will dissappear if DEBUG is not defined. Instead of having to brace the printf with #ifdef DEBUG/#endif in the code every time it is used the DEBUG0 can be used directly in place of the printf.

Here we come across a difference between macros and functions that requires some thought. The printf function can take a variable number of arguments but the above macro can only take one argument so it is only any good for printing constant strings. The only way around this is to have a range of DEBUG macros that accept different numbers of arguments. Here is one that will accept a printf format string and two variable items to print:

```
DEBUG3(S, X, Y) {printf("%s(%d) ",__FILE__, __LINE__); \
             printf(S, X, Y); }
```

A good convention is to end the name of the macro with the number of argumets it expects.

Another common way of debugging is to put an if statement around the printf so that the message is only printed if a condition is true. This can also be catered for neatly as a simple macro - just a variant of the DEBUG macro set discussed above. Here is a the DEBUG3 macro with an added if part:

```
IFDEBUG3(C, S, X, Y) if(C){printf("%s(%d) ",__FILE__, __LINE__); \
             printf(S, X, Y); }
```

The message is only printed if condition C is true. C can be anything that would be legal in an if statement.

```
      IFDEBUUG3(x > 10 && y > 20, "x is %d and y is %d\n", x, y);
```

Sensible use of macros can do much to make C programming easier to do and C programs easier to read. If debugs only take one line instead of 3 or more then they detract much less from the readability of the underlying program.

COPYRIGHT & THINGS

C Vu is composed entirely on an Atari 1040STf using Timeworks Desktop Publisher ST, and a combination of Word Perfect (which isn't), ST Writer Elite, 1st Word and the Megamax Laser Editor.

Thanks for this issue must go to all those who made the effort to write something for it and to Gollner Publishing for the printing.

Additional thanks to:

HiSoft, Metacomco, Electric Distribution, AS&T and Computer Manuals for taking the flyers at the Atari Show!

Also to David Eastwood at Sahara and...

Dave Hurley at Analytical Engines and finally....

.... to Rupert Steele for the frequent plugs in Personal Computer World (The End Zone)

Phil Stubbington
Editor