PIPELINES

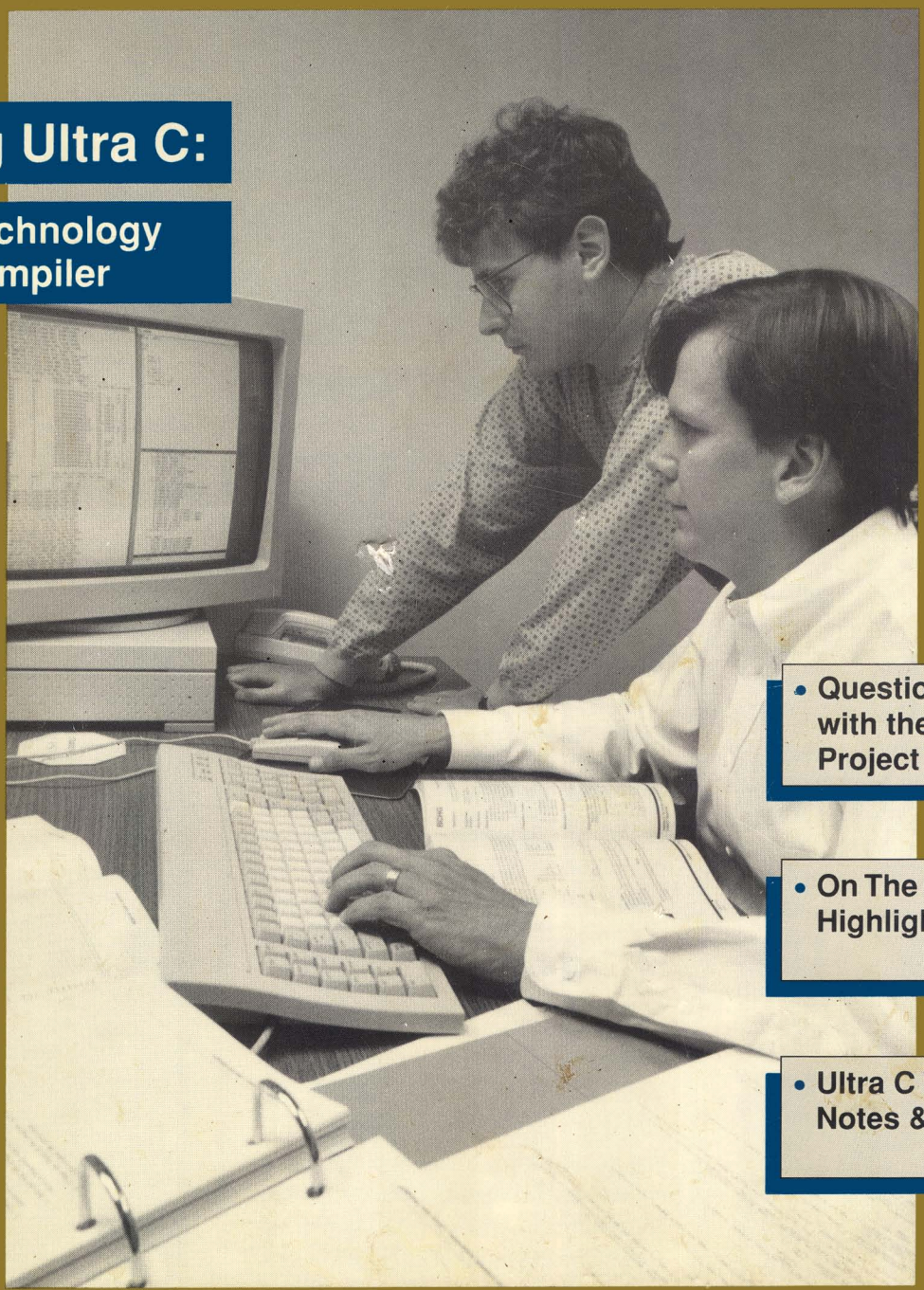
Volume 7 Number 3

Covering Microware's Real-Time System Solutions

Special ANSI C Issue

Introducing Ultra C:

Advanced Technology ANSI C Compiler



microware®

PIPELINES

Special ANSI C Issue
Volume 7 Number 3

PIPELINES is published quarterly by:
MICROWARE SYSTEMS CORPORATION
1900 NW 114th Street
Des Moines, Iowa 50325-7077
(515) 224-1929

Publication Coordinator:
David F. Davis

Editor:
Steve Simpson

Art Director:
Polly Steele

Photography:
David F. Davis

Microware Contributors:
Richard Russell, Ric Yeates, Rob Beaver, James Jones, Larry Crane, Steve Johnson, Ellen Grant

Table of Contents

On The Cover

MICROWARE'S RICHARD RUSSELL (SEATED) and Rob Beaver discuss development of Ultra C, Microware's new ANSI C compiler.

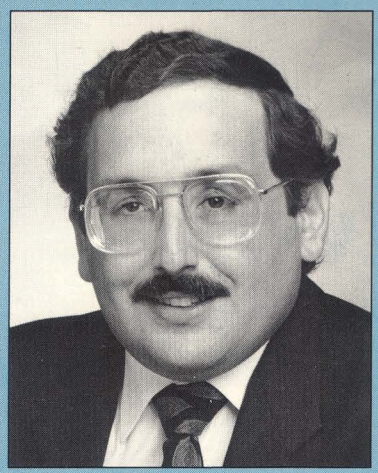
About This Issue

THIS SPECIAL ISSUE OF *PIPELINES* IS SOLEly dedicated to providing our readers with an overview of Ultra C. Most of the articles in this issue were written by the compiler design team to provide useful technical information about Microware's new compiler.

This is the first of many such special issues that will give readers an insightful look into the development and features of new Microware products.

In addition to new product issues, watch for an upcoming special issue dedicated to examples of OS-9 and OS-9000 applications from around the world.

Microware's New Compiler Technology



Ken Kaplan,
Microware's President

WHEN MICROWARE DEVELOPED ITS FIRST advanced language compiler back in 1978, I never thought I'd have the privilege of announcing another compiler that broke technological ground. Now I'm pleased to introduce Ultra C, Microware's new ANSI C compiler. This new compiler system incorporates revolutionary concepts in compiler technology including state-of-the-art optimizations, easy retargetability and simple portability.

I'm proud of the work that our compiler design team has done. They've added features that were developed from the latest trade and academic research. The optimization capabilities of Ultra C will

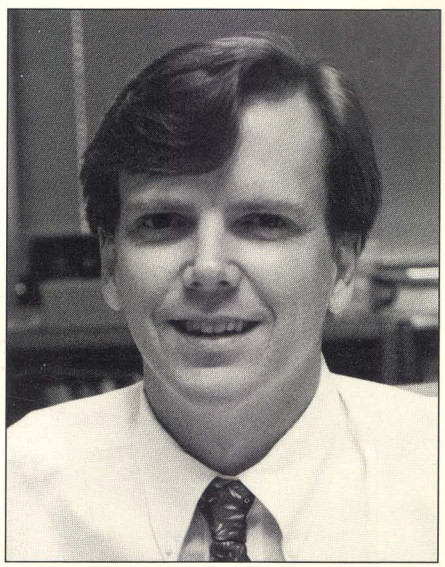
NEW PRODUCT

Introducing Ultra C
The Advanced Technology Compiler from Microware

MICROWARE IS PROUD TO INTRODUCE ***Ultra C****, the next generation of advanced compiler technology. Ultra C is an ANSI C compiler that incorporates revolutionary compiler architecture and state-of-the-art optimization techniques to extract ultimate performance for real-time applications.*

Ultra C has been rigorously tested and validated with the Plum Hall ANSI C Validation Suite to assure quality and ANSI compliance. And since Ultra C and its supporting tools are portable, users can easily migrate between existing Motorola 68XXX and Intel 80X86 microprocessors, as well as future microprocessor designs. Together with Microware's real-time operating systems, Ultra C brings new, industry-leading solutions for real-time applications.

by Richard Russell
Microware Systems Corporation



help programmers create code that is very small and very fast.

I'm equally proud of the extensive testing we've put Ultra C through. Before Ultra C was released, the compiler had successfully passed the massive Plum Hall ANSI C Validation Suite, followed by rigorous in-house testing and comprehensive beta testing around the world. Users will quickly recognize the high integrity and reliability this new compiler offers.

It's important to point out here that compilers aren't a sideline at Microware. We've been developing language compilers since 1978. All our compilers have been specifically designed to maximize the performance capabilities of the OS-9 and OS-9000 Real-Time Operating Systems. The release of Ultra C represents the culmination of 15 years as the leader in real-time system software. And, I think Ultra C will have a profound effect on the embedded real-time marketplace.

So, where do we go from here? The introduction of Ultra C is just the beginning. We've designed this compiler in such a way that we can easily target new microprocessors. We're looking beyond 16- and 32-bit CISC and RISC designs to emerging 64-bit architectures. We're also looking toward languages beyond ANSI C. We're investigating optimizations to further enhance application performance.

In Ultra C, users have a compiler that can enhance their current applications' performance today. More importantly, Ultra C provides a powerful pathway to future languages and target processors.

—Ken Kaplan

WITH THE PASSAGE OF THE ANSI C STANDARD, the development of our portable operating system OS-9000, and the limitations of Microware's existing compiler, it was clear that Microware needed a new C compiler system. This new compiler had to meet the following requirements:

- Provide ANSI C compliance
- Be easily retargetable to new processors
- Be highly-optimizing
- Integrate well into our operating system development and run-time environments
- Provide support for programming languages other than C
- Be portable to facilitate creation of cross compilers
- Pass quality assurance standards set by commercial standard compiler test suites

Microware faced several options for a new compiler, including third-party compilers, hybrids of in-house compiler components and third-party compilers, or creating an in-house compiler. After considering these solutions, it was decided that the way we could give our customers the best compiler for real-time environments was to design and develop a new compiler.

The design of such a compiler had to be one that would enable our customers to benefit from the latest compiler and microprocessor technologies, build on our experience with previous OS-9 and OS-9000 compilers, and meet the requirements as detailed previously.

Meeting the Requirements

As the project took shape, the compiler design team identified various components that could meet the project requirements:

- To make the compiler ANSI compliant we designed a new language front end and ANSI C libraries.
- To ensure compliance we acquired the Plum Hall Validation Suite and used the ANSI conformance component.
- To make the compiler retargetable we designed the front end to produce a machine-independent intermediate code, or I-code, so that this component would not have to be retargeted. We designed tools to make the job of retargeting the machine-dependent components easier.
- To make the new compiler highly optimizing we incorporated optimization algorithms from the latest conference proceedings, academic research and industry information.
- To facilitate creation of cross compilers, we implemented the compiler in carefully coded portable C.

Advanced Technology Compiler
Please turn to Page Nine

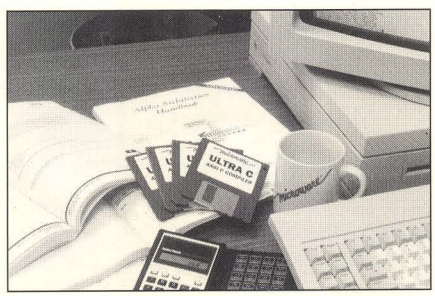
How to Get Ultra C

ULTRA C IS PACKAGED AS UPGRADES TO existing Professional OS-9 Version 2.4, OS-9000 Version 1.3 Development Paks and Version 3.2 of Microware's C Compiler. The upgrade package includes the Ultra C ANSI compiler distribution on magnetic tape or diskette, a new 800-page *Using Ultra C* manual and a 90-day "Hotline" Support Card.

The Ultra C upgrade provides OS-9 and OS-9000 users a logical path to migrate applications into the Ultra C ANSI environment. Ultra C easily installs on OS-9 and OS-9000 systems allowing immediate C development while still preserving access to Version 3.2 of Microware's C Compiler. Ultra C works with existing C V3.2 source code files and makefiles through the use of its tri-modal executive. Users can recompile existing applications without source code modification to improve speed and overall system performance.

Microware's **C Source Level Debugger (SrcDbg)** has also been updated with the release of Ultra C. The new SrcDbg is 100% ANSI compatible and accepts both Ultra C and C Version 3.2 source code, symbol tables and debug files.

Ultra C upgrades and ANSI C Source Level Debugger updates can be ordered directly from Microware or authorized Microware distributors, including OS-9 and OS-9000 system hardware suppliers. Stand-alone copies of Ultra C for additional OS-9 or OS-9000 machines are available. Contact Microware for availability of UNIX- and DOS-hosted cross versions of Ultra C. MSC



Ultra C, Microware's advanced technology compiler.

Questions and Answers with the Ultra C Project Team

Ultra C Project Team members are (back row, l. to r.): Larry Crane, Vice President of Advanced Research; Gwenna Jacobsen, cross development; Ric Yeates, compiler design; and Richard Russell, Director of Languages and Development Environments. Front row (l. to r.): Dennis Gabler, cross development; Rob Beaver, compiler design; Ellen Grant, technical documentation; James Jones, compiler design and Dave Lyons, debugging tools.

Q: *What is Ultra C?*

A: Ultra C is the new advanced technology ANSI C compiler from Microware targeted for Microware's real-time operating systems. It provides programmers with a high-performance compiler that incorporates state-of-the-art optimization strategies to produce fast, tight code for real-time applications.

Q: *What makes Ultra C unique compared to other commercially available compilers?*

A: Ultra C compiles all files into intermediate code (I-code). I-code representations of application code, C libraries and OS libraries can then be linked into a single I-code file that can then be passed through the I-code optimizer. This approach provides the opportunity to optimize at an interprocedural level, compared to older compilers that only perform at local or global optimization levels.

By optimizing the entire linked application, Ultra C allows users to perform span-dependent instruction optimizations, in-lining and reorganization of data areas for procedures relative to the application. This differs from older compilers that link after optimization in that older compilers prevent or minimize global optimization.

Q: *What resources did you tap for information about advanced compiler designs?*

A: We've searched through the latest conference proceedings, academic research and industry information for state-of-the-art compiler technology. It's important to understand that, because of Ultra C's architecture, we can easily implement new features in the future.

Q: *What is the intermediate code, or I-code, used by Ultra C?*

A: I-code is a proprietary binary representation of code that is source language-independent and target processor-independent. This means that Ultra C can easily have multiple language front ends, as well as multiple target back ends. The I-code level also provides a convenient place for optimizations since these optimizations benefit all languages and target processors.

Q: *How does Ultra C integrate with existing OS-9 and OS-9000 development systems?*

A: Ultra C will easily install on any Professional OS-9 Version 2.4 or OS-9000 Version 1.3 system. Ultra C offers compatibility with Version 3.2 of Microware's C compiler. Ultra C's system call libraries are designed to be complete and consistent so that system state code (including device drivers) are easier than ever to write.

Ultra C provides three options for compilation: backward compatible with Version 3.2, strict ANSI mode and an ANSI-extended mode. In the backward compatible mode, applications originally compiled under the Version 3.2 C Compiler will easily compile under Ultra C.

Q: *Does my existing Source Level Debugger work with Ultra C?*

A: No, Microware is releasing an ANSI C Source Level Debugger (SrcDbg). This new version of SrcDbg accepts both Ultra C and Microware C Version 3.2 source code, symbol tables and debug files.

Q: *Will Ultra C be available on cross-hosted development platforms?*

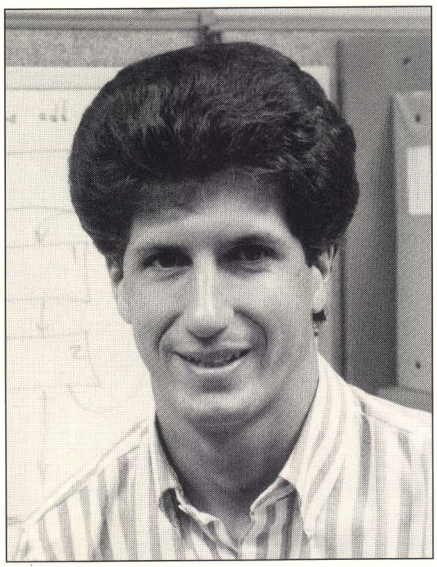
A: Absolutely. We designed Ultra C to be easily portable to platforms other than OS-9 and OS-9000 by writing the compiler in C. The entire compiler can be recompiled on the new host to target OS-9 and OS-9000 systems.

Q: *Besides 68XXX and 80X86, what other microprocessor will Ultra C target?*

A: Ultra C's enabling design will allow Microware to readily target emerging processor technology including RISC, 64-bit designs, VLIW, superscalar and super-pipelined architectures. MSC

Optimization Strategies with Ultra C

By Ric Yeates
Microware Systems Corporation



ULTRA C'S OPTIMIZATIONS ARE DESIGNED TO increase the execution speed and reduce the size of the final object code. Some optimizations sacrifice code space to increase execution speed.

Various compiler phases perform optimizations, each with their own emphasis. The front end is primarily responsible for translation-type optimizations such as constant folding and loop rotation. The I-code optimizer, in which the majority of optimizations are performed, deals with all the "classic" machine- and language-independent optimizations. The back end's task is to translate the I-code into optimal assembly language for a particular processor. The assembly language optimizer accomplishes processor-dependent optimizations beyond the scope of the back end. The assembler shrinks some instructions to their smallest form. Since each phase acts as a filter to eliminate non-optimal constructions from the previous phase, the final result is a highly-optimized executable.

Front End Optimizations

- Loop Rotation — the reduction of the branching complexity associated with *do*, *while* and *for* loops.
- Constant Folding — reducing operators whose operands are constants to the result of the operator.

I-Code Optimizer

- Constant Propagation — changing variable references to constants if the variable is known to contain a constant value.
- Constant Folding — determining the span in a function that the value of a local variable is needed. This frees its storage when it is not in use for other local variables to use.
- Variable Lifetimes — adding information to the I-code to allow the back end to make the most efficient use of the target register set.
- Common Subexpression Elimination — eliminating recomputation of identical expressions.
- Pointer Tracking — examining the contents of pointer variables when they are dereferenced to make the best determination of their effect on common subexpressions.
- Useless Code Elimination — deleting useless assignments and the computation of the right-hand side until all useless assignments have been eliminated.
- Assignment Translation — changing normal assignments into more efficient assignment operators wherever possible.
- Code Motion and Combining — moving multiple copies of the same code to a common location.
- Loop Invariant Hoisting — relocating invariant code from the inside to the outside of a loop.
- Induction Variable Strength Reduction — replacing multiplications involving a loop induction variable to a series of additions.
- Loop Unrolling — making more copies of a loop body to reduce the overhead of having to increment and check the loop induction variable.
- Function In-lining — removing the call to a function and replacing it with a copy of the function.

Back End

- Dynamic Register Set — allocating registers for local variables that are different for each function.
- Register Coalescing — performing operations such that the result ends up where it is going to be needed.
- Register Coloring — determining the best use of registers for a function and putting the least used variables on the stack.
- Data Area Layout — putting the most frequently referenced global data objects in the fastest storage area.

Assembler

- Branch Shortening — reducing the instruction size on branch instruction when the distance to the destination is known to be within certain limits.
- PC-Relative Addressing Mode Shortening — reducing the instruction size for the PC-relative addressing mode when the label is within certain limits.

Ric Yeates is a senior software engineer at Microware. He was responsible for the I-code and assembly optimizers, as well as the user interface for Ultra C. During his five years at Microware, Ric has also been involved in the development of OS-9000 and was a training and education instructor. MSC

ON THE C SIDE

ANSI C Highlights

by Ric Yeates
Microware Systems Corporation

MICROWARE'S NEW ANSI C COMPILER AFFORDS THE C PROGRAMMER with many more features than the Version 3.2 C Compiler. This article will highlight some of these features. For complete information, refer to the new *Using Ultra C* manual and a copy of the ANSI X3J11 standard. The features described in this article fall into the following four categories:

- variable type qualifiers
- function prototypes
- variable argument functions
- library routines

Variable Type Qualifiers

ANSI C adds two type qualifiers to the C language:

- ***const***
- ***volatile***

const Qualifier

const denotes that the storage for the object may not be modified. In other words, if you declare a variable to be a ***const*** qualified type, the compiler will refuse to generate code that modifies the storage.

The compiler automatically allocates storage for ***const*** qualified objects in the code space of the module. This allows large static tables to be stored in the module itself that is shared by all processes using the module, rather than in the data space of each process. This feature conserves memory.

For example:

```
const int c = 0xa110ca7d,     /* integer constant */
          *ptr = &c;          /* pointer to integer constant */
func()
{
   c = 10;              /* illegal: cannot change c */
   *ptr = 10;           /* illegal: indirection would cause
                           c to change */
}
```

volatile Qualifier

volatile specifies that the storage for an object can change at any time and that assignments or references to the object explicitly placed in the program must remain. ***volatile*** is often used for hardware registers. This prevents the compiler from throwing away code that looks useless. For example:

```
extern const volatile int real_time_clock;
func()
{
   register int time1, time2, i;

   time1 = real_time_clock;
   for (i = 0; i < 1000000; i++)
       ;
   time2 = real_time_clock;
   printf("time = %d\n", time2 - time1);
}
```

If ***real_time_clock*** had not been declared as ***volatile***, the compiler could have decided to simply copy the value in ***time1*** into ***time2*** since they appear to be getting the same value. Since ***real_time_clock*** is ***volatile***, it generates correct code for the above function by reaccessing ***real_time_clock*** for the assignment into ***time2***.

Function Prototypes

Ultra C accepts function prototypes. Function prototypes declare the type and number of parameters that a function is to be passed. With this facility, the compiler can ensure the proper calling of prototyped functions. For example, the following code declares a function called ***func*** that returns an integer and takes two ***floats*** as its only arguments. If you passed integers as the arguments to ***func***, they would automatically be converted to ***float***.

```
int func(float x, float y);
```

This prototype illustrates another aspect of prototyped functions, the "usual" argument promotions are not performed at the call site. Normally, passing a float to another function causes it to be promoted to a double before it is passed. If the prototype shown above had been processed prior to the call, both float type arguments would have been passed as floats.

All ANSI functions have a prototype in an ANSI header file. Microware also provides function prototypes for system call bindings in appropriate header files. This is designed to reduce the number of programming errors related to function calls.

Variable Argument Functions

Functions that take a varying number and type of arguments can also be prototyped. The ellipsis (...) is used to denote 0 or more further parameters. For example, the prototype for ***printf*** is:

```
int printf(const char *format, ...);
```

This means that ***printf*** returns an integer and takes a format string and 0 or more further parameters. The number and type of the parameters is dictated by the escapes in the format string.

For example, if you wanted to write a generic function to return the largest integer in its list of parameters you might use:

```
#include <stdarg.h>
int max(int num, ...);
int max(int num, ...)
/* num = number of integer parameters passed */
{
    va_list vp;
    int max, next;

    if (num <= 0)
        error(1, "can't find maximum in a list of %d
        integers\n", num);

    va_start(vp, num);    /* initialize scan of arguments */
    /* start with max as first argument */
    max = va_arg(vp, int);
    while (--num) {       /* for each subsequent argument */
        next = va_arg(vp, int);   /* get next argument */
        /* larger? if so, it's current max */
        if (next > max)  max = next;
    }
    va_end(vp);  /* we're done with variable arguments */
    return max;
}
```

Library Routines

Ultra C has a large number of new library routines. There are both ANSI defined routines and new system call bindings.

ANSI Functions

signal()

signal() is used to catch various signals that the OS or other processes can send to your process. In OS-9 and OS-9000, processes may send one another signals to communicate (see the appropriate *OS-9* or *OS-9000 Technical Manual*). There are also a number of signals the OS can send to your process when certain exceptions occur.

signal() allows you to set up a handler routine, ignore or exit for a given signal value. For example, the following code causes your program to ignore the keyboard interrupt signal and call ***bus_error()*** if your process gets a bus error (segment violation).

```
extern void bus_error(int);
main()
{
    signal(SIGINT, SIG_IGN);
    signal(SIGSEGV, bus_error);
    .
    .
    .
}
```

vprintf()

vprintf() is much like ***printf***, except that only one argument — an initialized variable argument list — is passed after the format string. This function can be very useful for generic error printing routines. For example, one might write ***error()*** (called in ***max()*** example) as follows:

```
void error(int exit_status, const char *format, ...);

void error(int exit_status, const char *format, ...)
{
    va_list vp;

    if (format) {             /* pass a format string? */
        va_start(vp, format); /* initialize vp */
        vprintf(format, vp);  /* do the printf */
        va_end(vp);           /* deinitialize vp */
    }
    exit(exit_status);
}
```

tmpfile()

The ANSI function ***tmpfile()*** can be used to automatically create a unique file. This file will be deleted when it's closed or when the program exits via the ***exit()*** function. It returns a pointer to a ***FILE*** structure suitable for high-level reading and writing.

atexit()

The ANSI function ***atexit()*** is used to "register" functions to be called when a program terminates via the ***exit()*** function. These functions could do general housekeeping such as unlinking from modules or events. Using this mechanism you could exit cleanly from anywhere in an application. You can "register" up to 32 function with ***atexit()***.

System Call Bindings

C functions for OS-9 system calls have been added to the standard libraries. These functions begin with ***_os*** or ***_os9***. They provide a consistent means to make system calls and receive return values. Each system call binding returns 0 if successful or the error number if not. Most bindings take pointers to areas that are filled with return values. For example:

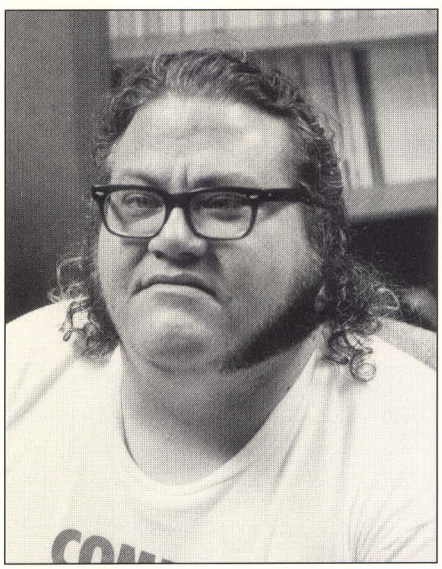
```
#include <const.h>
#include <modes.h>

/* In modes.h:
   "error_code _os_open(char *, u_int32, path_id *);" */

path_id open_for_read(name)
char *name;
{
    path_id retpath;
    error_code err;

    if ((err = _os_open(name, S_IREAD, &retpath)) != SUCCESS)
        error(1, "can't open %s for read - %s\n",
        name, strerror(err));
    return retpath;
}
```

Another feature of the system call bindings is that, in many cases, they are compatible with the OS-9000 system call bindings. This facilitates porting code from OS-9 to OS-9000 and vice versa.

Refer to the new *Using Ultra C* manual for more information on these functions and many more. MSC

ANSI C: Standard for a Common Language

By James Jones
Microware Systems Corporation



WHY DID THE AMERICAN NATIONAL STANdards Institute (ANSI) bother to create a C standard? There are several reasons. The de facto standard, Kernighan and Ritchie's *The C Programming Language* (called "K&R" henceforth), is vague in some areas. Existing compilers aren't necessarily good de facto standards, since nothing stops different implementers from interpreting ambiguities differently. Also, an implementation may even deviate from K&R—at least two discrepancies between a commonly-used C preprocessor and K&R have become "features" that many applications count on.

Also, the language has changed in the years since K&R was written. A strict K&R compiler, with a single name space for structure and union members and without any unsigned types save ***unsigned int***, would probably strike many C programmers today as intolerable. These extensions, whether linguistic or library, provide yet another chance for compilers to differ and make programs less portable. Variance between compilers leads to questions like: Which does your library have, ***index()*** or ***strchr()***? Are you sure you haven't picked a variable name that happens to be the same as that of some important library routine that standard I/O counts on?

For these reasons, and others such as internationalization and catering to optimizing compilers, the ANSI X3J11 committee developed a standard for C.

Formalizing the C Language

ANSI C formally defines some common C linguistic and library extensions or, in the cases of the aforementioned preprocessor features, provides a well-defined way to achieve the same effect. It explicitly states the stages that an "abstract compiler" and "abstract machine" should go through to, respectively, compile and execute a C program.

In theory, then, a user can sit down with the ANSI C standard and information specific to a particular compiler, and answer any questions that might arise about the behavior of the compiler when faced with a program. Further, assuming the program conforms to the standard, the user can answer any reasonable question about how it should behave when compiled and run.

In practice, an actual compiler and machine can take shortcuts as long as the implementers guarantee the results to be the same as if the abstract behavior were adhered to. For example, a C interpreter might perform various phases of compilation in a single program and not have a separate preprocessor, or a compiler may decide not to actually do the "usual arithmetic promotions" if the results are unchanged thereby.

New Features in ANSI C

ANSI C also adds several new features including:

- Explicit quoting and "token pasting" operations to build up new tokens from old.
- The operands of ***#include*** and ***#line*** directives may now contain text subject to preprocessing (e.g., macro expansion).
- "Function prototypes" so that references to functions other than their definitions can describe the number and types of their parameters. This can add some of the checking formerly done by separate programs such as ***lint***.
- A new specification for optional arguments is part of the standard, avoiding the old tricks used to implement optional arguments and giving the compiler a chance at more efficient calling conventions.
- ANSI added the "type qualifiers" ***const*** and ***volatile*** to allow for improved optimization while keeping optimizations out where the compiler could not otherwise tell they'd break one's code.
- Library functions have been standardized and significantly extended, and header files are now explicitly required to be safe for repeated inclusion.
- The open-ended ***locale*** library routines, along with "wide characters," "multibyte characters" and routines that manipulate them, support "internationalization." Internationalization allows for writing code to sell in varying markets with different languages, character sets and monetary units.
- Standard I/O in ANSI C has an improved way to specify buffering, and even looks ahead to mass storage of a size that may not be representable in a long int.

James Jones is a senior software engineer at Microware. His work on the new compiler included creating the ANSI-conformant front end and code generation methods of the back end. James has been with Microware for six years. MSC

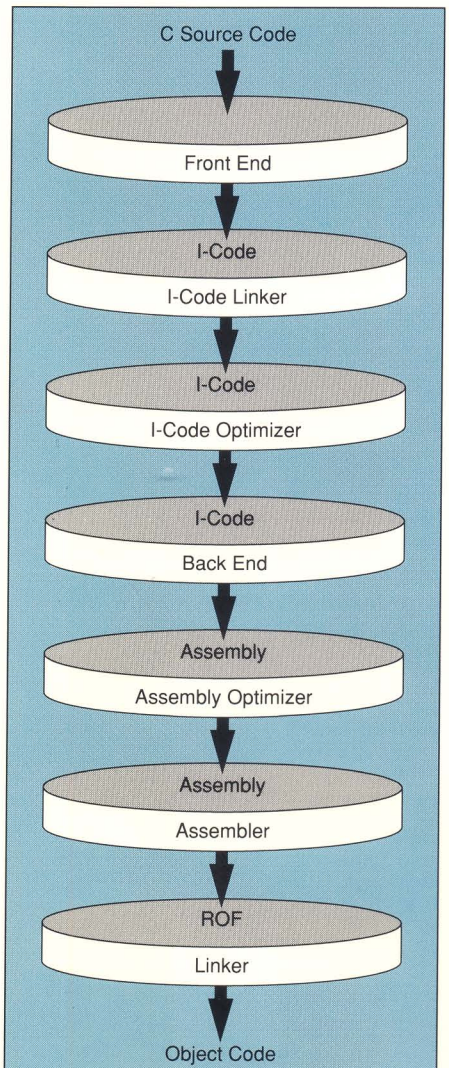
Continued from Page Three

- To ensure high quality we used the testing component of the Plum Hall suite. Then, we employed extensive alpha and beta testing programs.

To complete the design, we planned for the future in all areas of the new compiler so that it could be continually upgraded with new and better optimizations, more features to benefit the operating systems, and fitted with the newest programming languages.

Overview of Ultra C's Design

In general, Ultra C has seven major components as shown in the illustration below. It's important to understand these components and their individual functions to fully understand the capabilities available with Ultra C.



Ultra C compiles a single source file as shown above.

The Language Front End

The front end translates the source language to intermediate code. For C language, we developed an ANSI C front end to perform this translation phase.

The intermediate code, or I-code, that the front end produces is a machine-independent and source language-independent binary representation of the source file. The use of I-code allows the language front ends and target back ends to remain fully independent. In this way, enhancements can be added to the compiler system without writing a new compiler for each language or target processor.

Intermediate Code Linker

The intermediate code linker (or ***ilink***) functions like object code linkers with which most people are familiar. However, instead of producing an object code representation of the program, it produces an I-code representation. ***ilink*** allows partial linking of a program, as well as full linking including I-code libraries. ***ilink*** is also the builder of I-code libraries.

I-Code Optimizer

The I-code optimizer is a language- and machine-independent optimizer. ***iopt***, as it is called, applies optimizations on the following levels: local (within straight line code), global (across straight line code but within a function) and interprocedural (across functions). Since the optimizer is language- and machine-independent, any improvements or new optimizations automatically benefit all source languages and all target processors. Since ***ilink*** can provide an intermediate code representation of the whole program (libraries and all), ***iopt*** can "see" all functions and data within the program and optimize it as a whole. This is a distinct advantage over compilers which can only optimize one function or one file at a time.

Target Back End

The back end translates I-code into the target assembly language. The job of the back end is to lay out the data area, select the code to be generated according to time and space knobs, assign registers according to the greatest need, and, finally, generate code. Again, since we can represent an entire program in I-code, the back end can generate code which uses the most efficient methods to access global data because it knows the final offset the object linker will assign.

Assembly Optimizer

The assembly language optimizer performs machine-specific optimizations.

A Compiler from the Real-Time Experts

ULTRA C IS PART OF MICROWARE'S commitment to providing a single-source solution for real-time developers. By tightly coupling compiler development with our real-time operating systems, Ultra C offers users distinct advantages over third-party compilers. By maintaining development paths for both the compiler and operating systems, users are assured that they have the latest technology available.

Microware can also extend C and other languages to exploit the features specific to Microware's real-time operating systems. These extensions include support for storing constant objects in the code area, shared libraries, subroutine modules and trap handlers. In addition, other compilers can't or won't support features of Microware's real-time operating systems, particularly the generation of position-independent, reentrant code that is required by OS-9 and OS-9000.

By providing a "one-stop" solution for a compiler and OS, users know where to turn for technical support. When using a third-party compiler, users are faced with the issue of whom to contact with questions. And, shareware or public domain compilers offer little chance for technical help. With Ultra C and Microware's real-time operating systems, you can count on one point of contact for total support.

These optimizations are fairly trivial given the optimizations previously performed by ***iopt*** and in the back end. The assembly optimizer "cleans up" sequences of code which the back end cannot do, such as merging duplicated sections of code into one common section.

The Assembler

The assembler performs the traditional translation of assembly language to machine language. A new feature of the 680X0 assembler is that there is now one assembler to handle all processor family members. An option is used to specify the member for which it is assembling. Also, the assembler can now perform span-dependent optimizations. This means that when the assembler "sees" the program as a whole (because it had been previously linked at the I-code level) it can produce code in which all code references are of the optimal size. This greatly increases speed and decreases size on large applications.

The Object Code Linker and Librarian

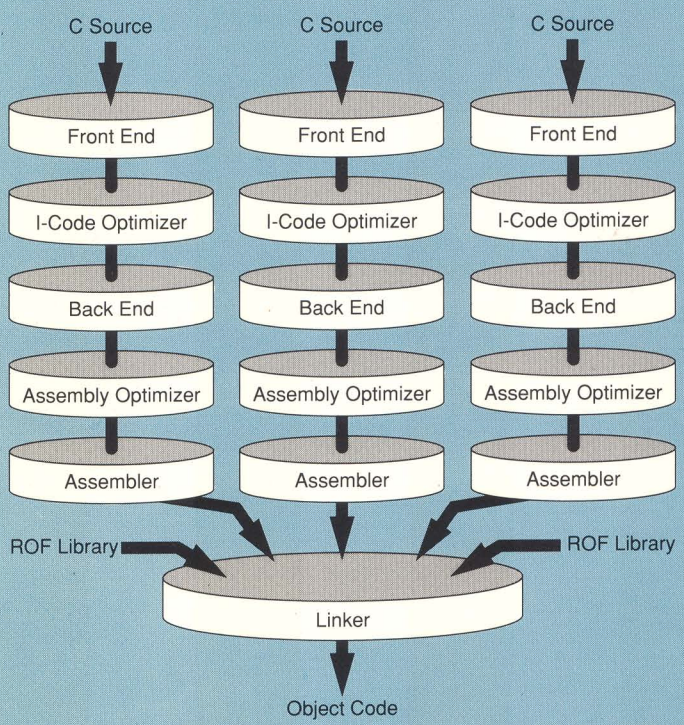
The object code linker performs the traditional linking of program object code and library object code into the final executable module. A new feature of the linker is that it now accepts a new type of library format; a pre-linked library built by a librarian utility called ***libgen***. ***libgen*** takes relocatables and builds a "pre-linked" library with a table of contents. ***libgen*** can create libraries from existing old-style libraries as well. ***libgen*** also has an option to display the contents of a new formal library. Given this new feature, the linker now does its job much more efficiently. We have seen increases in link speed of up to six times using the new libraries. The librarian utility solves the problems associated with library ordering found when trying to create large old-style libraries since the librarian automatically orders the libraries. The linker is backward compatible to the extent that it accepts both old-style and new-style libraries.

New Libraries

The new libraries which come with Ultra C are ***clib.l***, ***os_lib.l*** and ***sys_clib.l***.

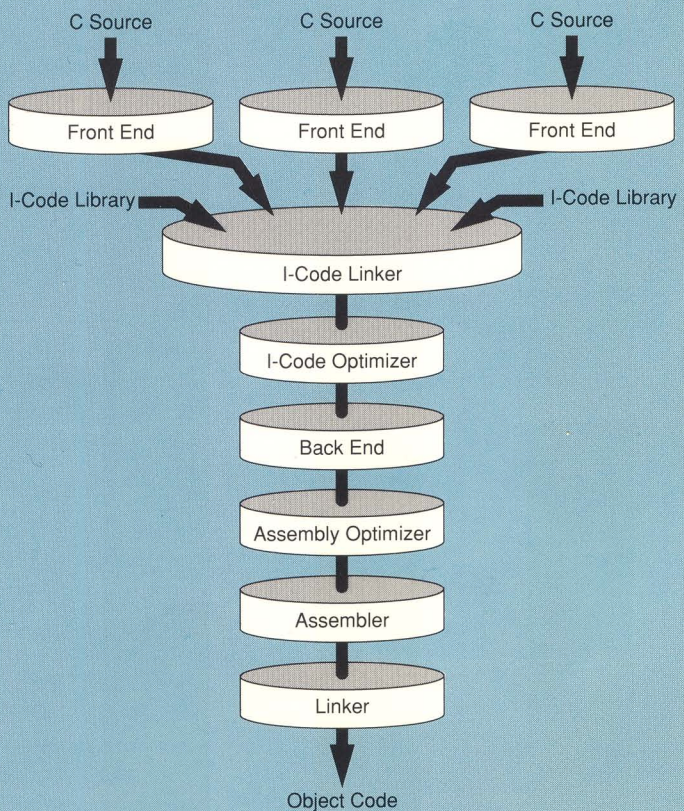
clib.l contains all the C language functions as defined by the ANSI C specification. ***clib.l*** has been shown to be conformant to the ANSI standard using the library test part of the Plum Hall Validation Suite. ***clib.l*** has been almost totally

Two Linking Options with Ultra C



Object Link Method — Each source file is compiled into its ROF (relocatable object file) and linked with other ROF files into an OS-9 or OS-9000 module. This method produces quality code with shorter compilation time. Each source file is fully optimized, but source files are not optimized with respect to one another.

Intermediate Code Link Method — Each source file is compiled to its I-code file which is linked with other I-code files and processed into an OS-9 or OS-9000 module. I-code libraries can also be linked at the



I-Code Linker. This method produces high-quality code. At each step after the I-Code Linker, Ultra C knows what each source file contains. This allows for greater optimization. However, total compilation time increases because the compiler re-optimizes every file during each compile time, even if changes are made to a single source file.

rewritten. The C high-level I/O functions have been improved to such an extent that using the OS's I/O system calls directly provides almost no advantage and, in most cases, the high-level I/O is much more efficient. This is because of improved buffering schemes and in-line macros.

os_lib.l contains C language level system call functions for OS-9 and OS-9000. This library contains all the functions available in the Kernel in a consistent calling syntax modeled after the OS-9000 design. These functions are ANSI C compliant since their names do not "pollute" the C namespace. Now, a programmer does not need to write C bindings for system calls as ***os_lib.l*** provides access to all system calls and parameters.

sys_clib.l contains all functions that previously existed on OS-9 and OS-9000 but are not a part of the ANSI C specification. These functions are here for backward compatibility and are not linked automatically except in the compiler's "backward compatibility" mode. There are new equivalent ANSI C compliant functions available in ***clib.l*** and ***os_lib.l***.

Math Support

Ultra C now generates code which assumes (in the 680X0 family environment) that the 6888X math architecture is present. This means the compiler generates the same "in-line 6888X" code for the whole 6888X family. When math hardware is not present, OS-9 will have a math emulation module to emulate the 6888X math coprocessor. And when math hardware is present, the program runs most efficiently with in-line math instructions without recompilation. Ultra C also will "in-line" function calls to trigonometric functions with the actual 6888X instructions for the greatest efficiency.

Compatibility

Ultra C is compatible with Microware's C V3.2 compiler. This is accomplished using Ultra C's tri-modal executive. This executive is really like having three executives. The backward compatibility mode is used when one wishes compatibility with C V3.2. This mode "looks" exactly like the C V3.2 executive and provides C source code and makefile compatibility with C V3.2. **C89** mode is an executive modeled after the POSIX P1003.2/D11.2 draft for a program to compile ANSI C source. This mode provides a standard UNIX-like executive for porting UNIX makefiles or for general use. The last mode is ***UCC*** mode, which is brand new, and is our most powerful and flexible mode. This mode provides access to all the features of Ultra C.

Extensive Documentation

The new *Using Ultra C* manual contains 800 pages of documentation on Ultra C and the new libraries' 550+ functions. Also available is a new *Ultra C Quick Reference Guide* which documents the compiler, command line options and all the library functions.

New Source Level Debugger

Microware's C Source Level Debugger has been updated to be fully compatible with Ultra C and fully reflect the ANSI C standard. Knowledge of ANSI C's function prototypes, type qualifiers and semantics have been added. The C Source Level Debugger is fully backward compatible, so it will debug C 3.2 and older compiler-generated programs.

Testing and Validation

The Plum Hall ANSI C Validation Suite is used to guarantee conformance of the compiler to the ANSI C standard and for general testing of the compiler. The Plum Hall Test Suite is a collection of C programs for testing with each section building upon the previous sections. The *Conform* section tests for ANSI C conformance, or various levels between Kernighan & Ritchie and ANSI C. The standards that are tested are ANSI X3.159-1989 and ISO/IEC 9899:1990. The *Cover* section generates self-checking C code to check all permutations of operators and data types. The *Limits* section is used to determine the hard limits of the compiler, such as how many characters are significant in an identifier. Finally, since it is impossible to test all possible legal expressions, a sampling approach is taken. The *Stress* section uses generated self-checking C code which contains complex self-checking expressions.

Plum Hall has also been chosen for conformance testing of ISO/IEC 9899:1990 by the British Standards Institution (BSI) and the JMI Institute of Japan. This underlines the quality level of the test suite and the quality level of testing for Ultra C.

The Future

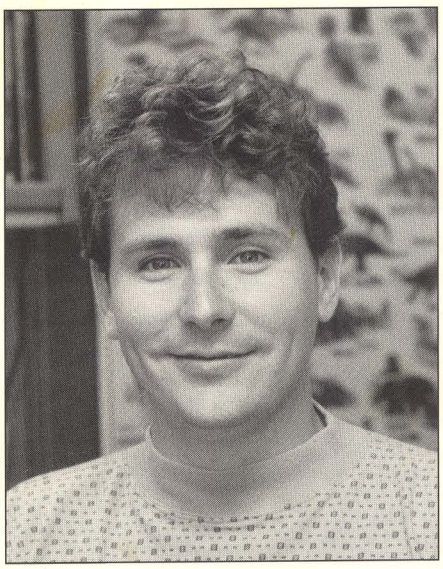
With Ultra C, Microware is offering a state-of-the-art compiler system. However, this is only the beginning and this release only reflects a small part of what is possible with this compiler design. Here are some of the areas we are already looking at for the future:

- Ability to debug optimized code: This helps cut down on compiling since there is no need to keep optimized and debugging versions of a program.
- Future Processors: Support for RISC, 64-bit processors, super scaler and VLIW.
- New optimizations: The main areas of emphasis in this area will be interprocedural optimizations. These include interprocedural register allocation, custom procedure calling conventions, global variables which are kept in registers across functions and instruction scheduling.
- Support for other programming languages such as C++.
- Further support for Microware's operating systems: Extensions include shared subroutine module language support.

Richard Russell is the director of languages and development environments for Microware and was the project leader for the Ultra C ANSI C compiler development. In addition to overall project design and management, Richard was specifically involved in the processor back end design and implementation. During his seven years at Microware, Richard has also held the position of manager of compiler development and was involved in the design and implementation of the C Source Level Debugger and UniBug, and the implementation of the ROM debugger.

Ultra C Technical Notes & Tips

by Rob Beaver
Microware Systems Corporation



MIGRATING TO MICROWARE'S NEW compiler will give users smaller, faster applications. Ultra C offers three options for compilation: backward compatibility with Version 3.2 of Microware's C Compiler, strict ANSI mode and an ANSI-extended mode that takes advantage of extensions by Microware. This article describes a compatibility issue that will be encountered even in backward compatibility mode, as well as a few tips to start your migration toward a strictly ANSI-conforming program.

Compatibility Issue With V3.2

Assembly language escapes are handled differently under the new compiler because of ANSI standard restrictions on preprocessor directive handling. The ***#asm*** and ***#endasm*** directives or the **@** sign will no longer work and will return error messages when code containing them is compiled under the new compiler. The new format is as follows:

```
_asm(<string constant>, [<expression>
        [, <expression> ] ])
```

The string constant may contain format escapes in the form %**<*n*>** where **<*n*>** is a decimal digit from 0 to 9, or %% which will generate a literal percent sign (%). The escape is replaced by the result of the **<*n*>**th expression, which must be a numerical constant. More than one %**<*n*>** with the same **<*n*>** may appear in the string constant and will be replaced each time with the same value.

Here is an example of 68000 Assembly code using the Assembly language escapes available under Ultra C:

```
_asm("Func: link a5,#0");
_asm(" move.l a0,-(sp)");
_asm(" move.l d(a6),a0");
_asm(" move.l %1(a0),d0", offset_of(d,
        int1));
_asm(" move.l (sp)+,a0");
_asm(" unlk a5");
_asm(" rts");
```

A utility, ***deasm***, is supplied with Ultra C which will convert these items from the old format to the new format.

Strictly-Conforming ANSI Programs

Ultra C provides an option that lets users compile source code in a strict ANSI compliant mode. A stepwise migration from K&R to fully-compliant ANSI code can be achieved fairly easily.

- Convert all references to non-ANSI functions to the equivalent ANSI function (e.g., replace calls to *index()* with calls to *strchr()*).

  Benefit: Your code can be migrated to ANY platform with an ANSI C compiler.
- Put prototypes in individual files for all functions declared in the file.

  Benefit: The compiler can type check the parameters you are passing to functions internal to the file.
- After trying the prototypes in the individual files move them to your header files where they will be globally seen.

  Benefit: The compiler can type check parameters for all calls to global functions in your application.
- If a library call is used, the corresponding header file must be included.

Rob Beaver is a senior software engineer at Microware. His areas of work on Microware's new compiler included the assembler, linker, librarian, libraries and floating point math support. Rob has been with Microware for four years. MSC

microware®
MICROWARE SYSTEMS CORPORATION
1900 N.W. 114th Street
Des Moines, Iowa 50325-7077

Bulk Rate
U.S. Postage
PAID
Des Moines, IA
Permit #2864