TEK INFORMATION FOR TECHNOLOGISTS

SEPTEMBER 1981

TECHNOLOGY report

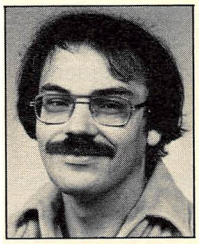
A (W H I V R X
D , G Y R I N B
P K U N ; G C
U H G T F R
P O I J) H . T E
M K J N H B
V F C D E W I
N G I W)

A PASCAL COMPILER FOR THE MOTOROLA 68000

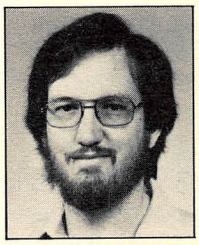
1	1			Ø	1
Ø	Ø	1	1	1	Ø
1	1	Ø	1	Ø	1
Ø	1	1	Ø	1	1
1	Ø	Ø	Ø	1	1
1	1	Ø	Ø	Ø	1
1	1	1	Ø	1	Ø
1	1	Ø	1	Ø	Ø

Tektronix®
COMMITTED TO EXCELLENCE

A PASCAL COMPILER FOR MOTOROLA 68000 FIRMWARE DEVELOPMENT



Allen Wirfs-Brock is a software design engineer in the Systems Engineering and Technology Group, part of the Design Automation Division (DAD). In the early 70s, Allen was a programmer in Data Processing. He has since worked on assemblers for the 8002, investigated interactive program environments in Tek Labs, was the project leader for GCS Pascal – he wrote the code generator, and now, in DAD, is investigating object-oriented programming languages. Allen was awarded a BS in computer science at the University of Oregon in 1977, culminating a two-year departure for that purpose.



Paul L. McCullough is a software design engineer in the Systems Engineering and Technology Group, part of the the Design Automation Division. Paul has been implementor or co-implementor of three Pascal compilers. He joined Tektronix four and a half years ago. Previously, Paul was involved in the design and implementation of operating systems and data base management systems at Burroughs Corporation. Paul is currently exploring software engineering environments; in particular, object-oriented programming systems.

Background

Pascal is a computer programming language known for its unique combination of simplicity, power, portability, and rigor. For several years, interest in using Pascal as a microprocessor systems implementation language has existed within Tektronix. However, the lack of high-quality Pascal compilers for commonly used microprocessors has limited the use of Pascal for the development of product firmware. Pascal interpreters (such as UCSD Pascal) have been available, but their performance has not been adequate for most firmware applications.

In early 1980, GCS engineering was about to start several firmware-intensive product development efforts employing the Motorola 68000 microprocessor. Because of the size and complexity of the projects, a high-level language was considered to be an essential implementation tool. Unfortunately the only high-level language available for the 68000 at that time (a Pascal compiler developed outside of Tektronix) was neither powerful nor reliable enough for Tek product firmware. For these reasons, GCS engineering chose to develop its own 68000 Pascal compiler.

Specification of the Language

Pascal was originally designed to be an instructional language for mainframe computers. For this reason, Pascal lacks several features that are generally considered essential for a microprocessor system implementation language. Such features include:

- separate compilation of Pascal procedures,
- the ability to call assembly language routines from Pascal,
- the ability to write interrupt service routines in Pascal,
- the association of Pascal variables with absolute memory locations (primarily to support memory-mapped input and output), and
- efficient manipulation of bit fields.

Many implementations of Pascal have attempted to correct these deficiencies via numerous extensions to the language (for example, some compilers extend Pascal variable declarations to include an absolute address specification). Such extensions often result in a plethora of special cases which destroy the elegant consistency of Pascal.

The design of GCS 68000 Pascal attempts to avoid such special cases. It implements the language as defined in the proposed international standard for Pascal. Standard Pascal is sufficently flexible so that a well designed implementation may support several systems-programming features without modifying the language definition. For example, Pascal subrange types may be used to declare unsigned, or "short" integer variables, and Pascal sets may be used to manipulate bits. GCS Pascal recognizes such special usages and attempts to generate optimal code for them.

The standard language was augmented with a small, consistent set of extensions to support systems programming. Minor extensions, which have been widely accepted by Pascal users, include non-decimal numeric constants and a default case statement alternative. The only major extension supports modular programming.

Modularity Features

GCS Pascal supports three forms of separately compilable modules, referred to as units. A *program unit* is a Pascal main program. An *interface unit* provides definitions of objects (constants, types, variables, procedures, and functions) that may be used within other units. An *implementation unit* implements the objects that are defined in an interface unit. An implementation unit can be written in either assembly language or Pascal.

The modularity features of GCS Pascal are quite powerful. The runtime routines that handle text input/output (that is, reading integers or characters, or the writing of integers, strings, Booleans, and characters) for GCS Pascal are entirely written in GCS Pascal. As another example, several of the

users of the compiler have implemented device drivers in GCS Pascal.

Most of Pascal's system-programming deficiencies can be overcome by using these modularity features. Variables that are defined in interface units may be assigned to absolute addresses using an assembly-language implementation unit. A Pascal interface unit may use formal procedure parameters (a feature of standard Pascal) to define a procedure which, in turn, would be called to install Pascal procedures as interrupt handlers. This installation procedure – which could be as short as three 68000 instructions – would be implemented in an assembly-language implementation unit.

These extensions are a subset of those orginally suggested several years ago by a committee of Tek engineers from several business units. Although the language (sometimes referred to as "Tek Pascal") specified by that committee has never been implemented, it was a useful guide for the design of GCS Pascal.

The Compiler

The compiler for GCS Pascal consists of two programs. The first, often referred to as the *front-end*, is target-machine independent (that is, it is independent of the machine on which the user's Pascal program will be run). The front-end performs the syntactic and semantic analysis of Pascal programs. If a program is error-free, the front-end translates it into an internal representation that is suitable for processing by the second program.

The second program is the 68000 code generator. It reads the internal representation of a Pascal program and produces a file of 68000 assembly language instructions, which may then be assembled with a 68000 assembler to produce a 68000 object file. Since the code generator produces assembly language, rather than binary object files, the compiler can be used in environments with different operating systems and different object file formats.

To develop a compiler for another microprocessor, a new code generator that processes the internal representation could be constructed – *but the front-end would remain unchanged.* In fact, if several code generators were to be developed, a user could use the same internal representation to produce code for very different microprocessors. Having only one program that performs the syntactic and semantic analysis ensures compatibility of source programs – even on different microprocessors.

Both the front-end and the 68000 code generator are written in Pascal. The compiler is capable of compiling itself, thus allowing the compiler to be transported from the current host machine (a DECSYSTEM-20) to a 68000 computer.

The Front-End

The front-end has two primary responsibilities: (1) to determine whether the unit being compiled complies with the language specification, and if so, (2) to produce an internal representation of the unit for consideration by the code generator(s). Should the front-end detect syntactic or semantic errors in the compilation unit, it reports these discrepancies with meaningful error messages.

The front-end is a top-down, recursive-descent compiler to which the Pascal language naturally lends itself. The advantages of this compilation technique are many: the compiler

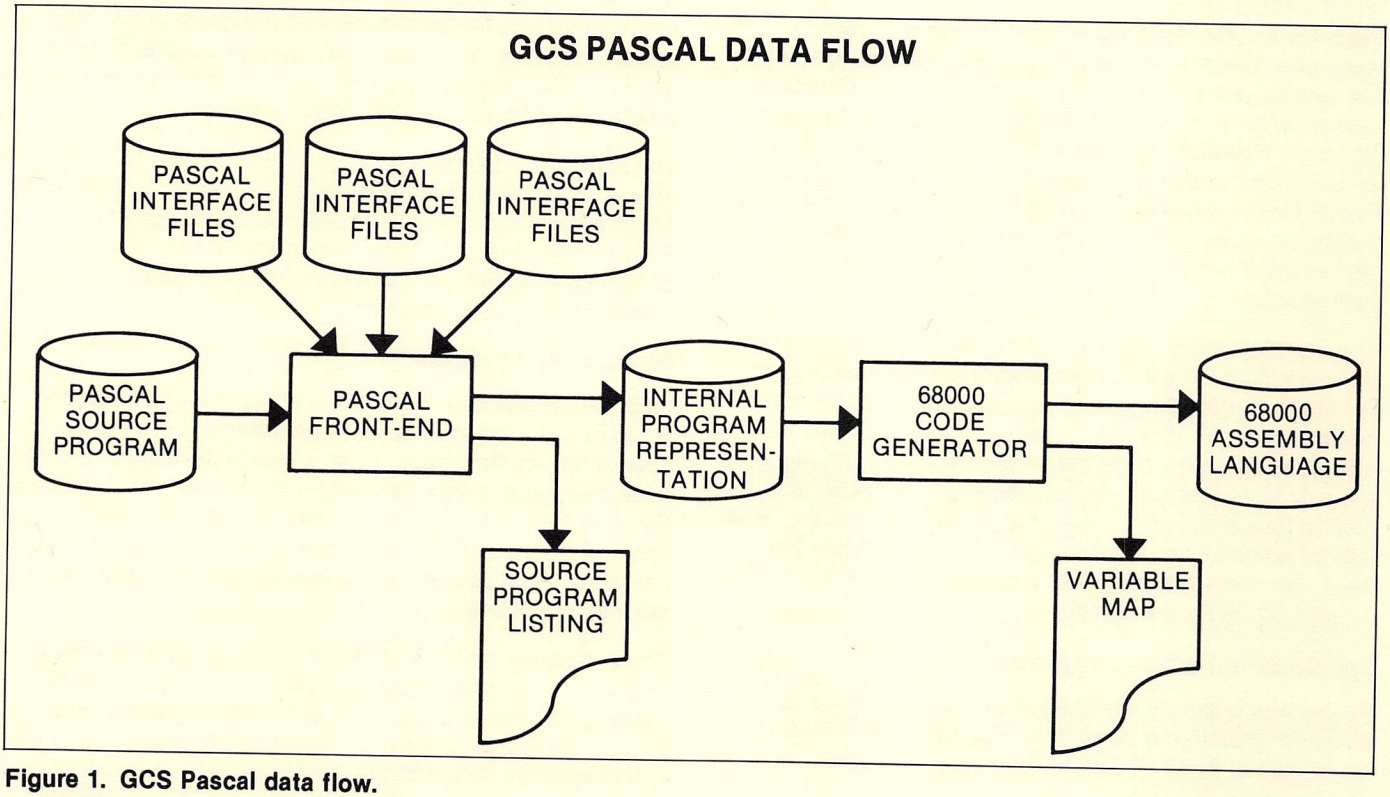


Figure 1. GCS Pascal data flow.

is simple, easily understood, readable by people, reliable, and easily maintained. Each procedure in the compiler has a specific task and is relatively short. As an example of the simplicity of the procedures, the routine that compiles Pascal *If* statements consists of less than 15 Pascal statements in its own right.

As mentioned previously, GCS Pascal compiles modules separately. The front-end reads the interface units needed by the module being compiled and makes available the identifiers exported by those interfaces.

The Internal Representation of Pascal Programs

Pascal programs comprise declarations and statements. Declarations provide information to the compiler about the labels, constants, types, variables, and procedures the software engineer will use in the program. Statements describe the actions and flow of control that will take place when the program is executed.

Because the front-end cannot make assumptions about the target machine, it has no knowledge of registers, stacks, addresses, or other artifacts of addressing in common machine architectures. Instead, the front-end assigns *item numbers* to each label, constant, type, variable, procedure, and function declared in the source program. These item numbers act as names or "handles" for the various declarations. Each statement of the source program is decomposed into one or more *quadruples*. A quadruple has an operation code and zero to three operands depending on the operation. Many operations yield temporary results; these temporary variables are also assigned items numbers. For clarity, in examples given in this article, we do not present the item numbers; we use the identifiers from the program text. For example, the Pascal statement:

a: = b + c + d;

where a, b, c, and d are all integer variables, would generate the following quadruples:

add b to c → temp1

add temp1 to d → temp2

move temp2 → a

The 68000 Code Generator

The 68000 Pascal code generator is a program that accepts as input the internal representation for a Pascal program and translates it into a 68000 assembly language program. A simple code generator would generate a fixed sequence of 68000 instructions for each of the different quadruples. Since the semantic level of the quadruples is closer to Pascal than to 68000 machine language, such a simple translation scheme would produce very poor 68000 code. GCS Pascal is intended for producing *product quality* code; that is, code which is efficient and reliable enough to be used as Tek product firmware. The code generator meets this goal by incorporating a number of code optimization techniques.

An optimizing code generator can have one of two goals: the minimization of the execution time of the generated program,

GCS PASCAL DATA TYPES

Pascal Type	Sample Declaration	Implementation
enumerations	type status =(notReady, Ready, Stopped);	1 byte 2 bytes if more than 256 elements

subranges	type byte = 0..255;
	word = 0..65535;
	signed_word =
	−32768..32767;
		unsigned 1 byte integer
		unsigned 2 byte integer
		signed 2 byte integer

integer	var i: integer;	32-bit signed integer
real	var pi: real;	IEEE 32-bit floating point
char	var c: char;	1 byte containing ASCII character
Boolean	var flag: Boolean;	1 byte containing 0 or 1
pointers	type ptr = ↑ byte;	4 byte field containing a 24-bit address

sets	type bits8 = set of 0..7;
	bits32 = set of 0..31;
	biggest_set = set of 0..2039;
		1 byte
		4 bytes
		255 bytes

Table 1. GCS Pascal data types.

or the minimization of the size of the generated program. These purposes are often in conflict. Since memory is still a scarce resource in many Tek products, GCS Pascal resolves this conflict in favor of minimum program size.

The complexity and code quality of a code generator depends upon how many quadruples must be simultaneously examined to perform an optimization. The simplest optimizations examine a single quadruple. More complex optimizations examine all the quadruples in a basic block (a sequence of quadruples with no intervening branches or labels). These first two classes of optimizations are commonly called local optimizations. The most complex class of optimizations (global optimizations) require the examination of the entire program. The majority of the optimizations performed by the GCS Pascal code generator are local optimizations. The highly structured nature of Pascal permits local optimizations to perform many code improvements that can only be performed by a global optimizer for other programming languages such as C. (C is a widely used programming lanquage.) A partial list of the optimizations includes:

- common subexpression elimination,
- constant folding,
- dead code elimination,
- copy propagation,
- deferred assignments, and
- span-dependent instruction optimization.

The ultimate purpose of any optimizing compiler is to produce code that approaches the quality of handwritten assembly language code. The success of a compiler in obtaining this goal is difficult to measure. The code written by a good programmer for a short program or program fragment is usually better than the code generated by a good optimizing compiler. The compiler's advantage is that it can optimize an entire large program – this may be very difficult for a human. GCS Pascal averages between nine and ten bytes of 68000 code per executable Pascal statement (in the 68000 architecture, the shortest instruction is two bytes, the longest is ten bytes).

The Pascal Execution-Time Environment

Pascal programs have been traditionally executed under the control of an operating system. The operating system normally provides support for certain features of the Pascal language, such as external files and dynamic storage allocation. Since GCS Pascal is intended for use as a system implementation language, the programs it generates must be independent of any particular operating system. In fact, these programs may not even assume the existence of an operating system. (GCS Pascal might be used to write the operating system).

Most GCS Pascal programs will operate independently of any external support. GCS Pascal programs automatically allocate storage for local and global variables. They are reentrant. A GCS Pascal program may call other GCS Pascal programs (including itself) as subroutines.

Code Generation Example

A Pascal code fragment:

```
const
   c = 3;
var
   x : array [0 .. 10] of 0..255;
   i : integer;
begin
   .
   .
   .
   x[i] := x[i] + 2*c;
   .
   .
   .
```

Quadruples produced by front-end:

```
index X by I → temp1
index X by I → temp2
multiply 2 by C → temp3
add temp2. indirect to temp3 → temp4
move temp4 → temp1. indirect
```

Quadruples after optimization:

```
index X by I → temp1
add 6 to temp1. indirect → temp1. indirect
```

Resulting 68000 code:

```
MOVE. L   I(A7), D0
ADDQ. B   #6,X(A7,D0)
```

Figure 2. Code generation example.

The code generated by the compiler does not directly implement dynamic storage management or files. Instead, code is generated to call support routines to perform these functions. The specification of GCS Pascal includes a precise definition of these routines' interfaces and functionality. The GCS Pascal support package includes a standard implementation of most of these routines, which should be satisfactory for most users. When they are not, users may provide their own routines that implement the support functions in a manner that is compatible with their particular system or application.

Project History and Current Status

Development of GCS Pascal was begun by the authors in March 1980. The first version of the compiler (which performed some optimizations but lacked files and floating point variables) was available for use in August 1980. A second version, which supported files, floating point variables, and many additional optimizations, was completed in March 1981.

The compiler is heavily used in GCS; two GCS products nearing release use it extensively. Thousands of compilations using the compiler are performed each month. It is also relatively error-free: only one bug (a very minor one) has been found in a recent two month period.

Available documentation includes specifications for the language, the execution-time environment, the internal representation, and internal documentation of both the front-end and the 68000 code generator.

GCS Pascal currently operates as a cross-compiler resident on DEC-10/20 computers. GCS Pascal has been used to compile itself to 68000 machine code, but has never operated on a 68000. Minimal effort (one to two person-months) would be required to install GCS Pascal on a 68000 or other 32-bit computer that has a Pascal compiler (such as a VAX).

Approximately 40 percent of the code generator portion of the GCS Pascal compiler is involved with the generation of 68000 machine code. The development of a code generator for another target machine would entail the replacement of that 40 percent of the code generator with code for the new target machine. A runtime support package for the new target machine would also be required. It would probably take between three and nine person-months to re-target the GCS Pascal compiler, depending upon the architecture of the target machine and the skill of the implementor(s).

For More Information

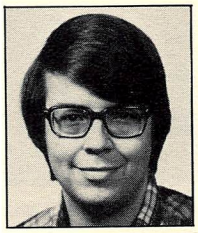
For information concerning the operation of GCS Pascal upon the DEC-10/20 computers in Wilsonville use the TOPS-10 HELP command:

HELP GCS:GCSPAS

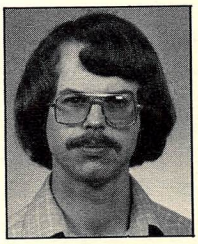
For other information concerning the GCS Pascal compiler, contact Allen Wirfs-Brock, ext. WR-1340. □

PATENT RECEIVED: No. 4,247,920

MEMORY ACCESS SYSTEM



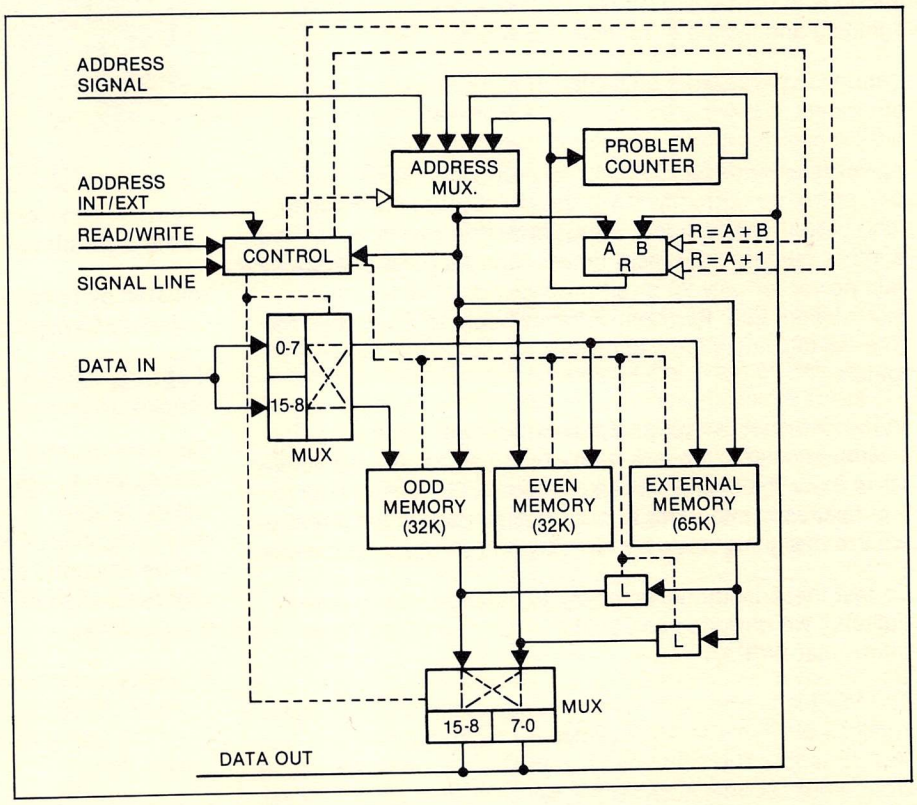
Richard Springer, GCS Mass Storage, W1-3535



John Theus, GCS Processor Development, W1-3573

Richard Springer and John Theus have received a patent for a means and a method for transferring information into and out of a memory system without regard to byte boundaries. With their invention, no pre- or post-processing is required.

With this invention, bytes A and B of a two-byte word can be stored in or retrieved from memory locations defined by N and N + 1 without regard to whether N is an odd or an even memory address. The invention maintains a preselected logical relationship between those two bytes as they enter or leave the memory system. For memory which is byte wide, as in ROM packs, the memory system generates two memory accesses to load or store in A and B.



The patent also covers the inclusion of the program counter (PC) in the memory system. Because the PC is part of the memory system:

- instructions are prefetched,
- instruction operands, which contain addresses, are used to automatically cycle the memory to retrieve the operand data, and
- branch operands, which are PC relative addresses, are automatically added to the PC.

All this is done without arithmetic logic unit (ALU) intervention.

This new method for handling memory and the use of a 16-bit slice ALU enabled the 4052 and 4054 Graphic Computing Systems to operate faster than their predecessors. The ALU and this memory system emulate a superset of the Motorola 6800 instruction set and thus did not require redesign of the 4050 series firmware. □