Advanced
Micro
Computers
A subsidiary of
Advanced Micro Devices

AmZ8002 C System Interface

User's Manual

	REVISION RECORD
REVISION	**DESCRIPTION**
01	Prelimiary Issue
(4/22/81)	
02	Manual Updated
(5/8/81)	
A	Manual Released
(5/29/81)	

Publication No.
059910610-001

REVISION LETTERS I, O, Q AND X ARE NOT USED

Address comments concerning
this manual to:

ADVANCED MICRO COMPUTERS

Publications Department
3340 Scott Boulevard
Santa Clara, CA 95051

PREFACE

This document covers the host- and target-dependent features of Advanced Micro Computer's C language compiler for the AmZ8002*, which is hosted under AMDOS. The document has five main parts:

1. Chapter 1 contains an overall guide to the C programming process, from source program (through compilation, assembly, and linking) to execution. All the AMC utility software used in the process is described. At each stage references are made to the relevant manuals for more details. Also included is an overview of the C language itself and descriptions of the host and target environments.

2. Chapter 2 contains a description of the assembly language code generated by the C compiler, including a discussion of certain dynamic run-time structures that are used by C programs (for instance, stacks). This information is useful in writing assembly-language subroutines to interface with C programs.

3. Chapter 3 describes the differences among the execution environments of the three principal system configurations for which C programs may be targeted.

4. Chapter 4 contains an annotated example, taken from source code through each step of the process.

5. Appendices A-E contain reference information on differences between AMC C and Standard UNIX** C, invocation procedures, the libraries, error messages, and differences between successive releases of the C Compiler product.

This manual should be used in conjunction with the following document:

- The C Programming Language, by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, 1978.

and the following AMC manuals:

MANUAL TITLE	PUBLICATIONS NUMBER
MACRO8000 Assembler User's Manual	00680119
LINK8000 Linker User's Manual	00680148
Am96/4016 Evaluation Board User's Manual	00680131
AMDOS Operating System User's Manual	059910425-001
RTE16/8050 Real-Time Emulator Support Package User's Manual	059910320-001

*Z8000 is a Trademark of Zilog, Inc.
**UNIX is a Trademark of Bell Telephone Laboratories

TABLE OF CONTENTS

D. THE RUN-TIME LIBRARY

E. ERROR MESSAGES

F. RELEASE INFORMATION

FIGURES

TABLES

INDEX

CHAPTER 1

HOW TO USE THE AmZ8002 C COMPILER

INTRODUCTION

The AmZ8002 C Compiler (ACC) translates programs written in the C programming language into AmZ8002 assembly language. The compiler itself executes on the AmSYS™8/8 under the AMDOS operating system. The C language is defined in The C Programming Language, by Brian W. Kernighan and Dennis M. Ritchie. (Differences between ACC and the standard definition of C are outlined in Appendix A.) AmZ8002 assembly language is defined in the AmZ8001/2 Processor Instruction Set, published by Advanced Micro Devices, and in the AMC document, The MACRO8000 Assembler User's Manual. AMDOS is described in the AMDOS Operating System User's Manual.

Assembly language modules generated by ACC can be translated by the AmZ8000 Assembler, MACRO8000, into corresponding modules of relocatable AmZ8000 object code. These relocatable modules, together with various AMC-supplied library functions can then be linked by the AmZ8000 Linker, LINK8000, into executable AmZ8000 binary files. (MACRO8000 is described in the MACRO8000 User's Manual; LINK8000 is described in the LINK8000 User's Manual.)

The HOST program allows binary files to be downloaded to the Am96/4016 Evaluation Board or the Am96/4116 MonoBoard Computer for execution. RTE16 software allows users to download executable object code to an RTE16/8050 real-time emulator system for testing and debugging. The programs PREHEAT, PPROG, and PLPROG are used to burn debugged programs into PROMs. (HOST is described in the Evaluation Board User's Manual PROM-burning software is described in the AMDOS User's Manual, and RTE16 software is described in the RTE16/8050 User's Manual.)

The remaining sections of this chapter describe the C language itself; the AMDOS development system environment and the AmZ8000 target environment; the programs that make up the AMC C Software Package (compiler, assembler, run-time library, and linker); and the sequence of steps users must follow to compile, assemble, link, and execute C programs.

OVERVIEW OF THE C LANGUAGE

The C programming language was originally developed at Bell Telephone Laboratories as a systems implementation language for the UNIX operating system. Virtually all of UNIX itself, as well as its utilities, compilers, and other support software, has been written in

C. The AMC C Compiler is compatible with Version 7 UNIX C (with a few exceptions described in Appendix A) and is designed to produce code for the AmZ8002 16-bit microprocessor.

C is a low-level programming language in the sense that important aspects of the hardware can be directly manipulated from within the language, but it includes many features of higher-level languages. For example, it supports a number of data types, and it offers structured control flow and a wide variety of operators.

DATA TYPES

C recognizes several elementary data types: bit fields, signed and unsigned characters (8 bits), signed and unsigned integers (16 bits), pointers (16 bits), signed and unsigned long integers (32 bits), single-precision floating point numbers (32 bits), and double-precision floating point numbers (64 bits). From these elementary data types more complex collections can be created: arrays of objects, each having the same data type; and structures (called records in some languages) of objects with arbitrary types.

The set of data types simultaneously covers all the hardware-supported data types of the AmZ8002 and also offers mechanisms for extension to more complex cases. C also provides a facility called **typedef** for creating new data type names; for example, using **typedef** COLOR can be defined as a data type. The names RED, GREEN, and YELLOW can then be declared as objects of the type COLOR.

OPERATORS

C provides an unusually rich set of operators to manipulate the elementary data types. These operators are listed in table 1-1.

C permits extensive manipulation of pointers, i.e. variables that contain the addresses of operands. In C, the unary operators * and & can be combined with the arithmetic operators to reference memory in as direct and efficient a manner as in assembly language programming.

FUNCTIONS AND PROGRAM ORGANIZATION

The basic organizational unit of C programs is the function. Every C program is a function in its own right, called **main**, and makes use of both predefined functions (such as **abs** -- find the absolute value of a number) and user-defined functions (which fill the same role as subroutines or procedures in other languages). Functions can be compiled independently and later linked together for execution. C functions can easily be made recursive and re-entrant.

TABLE 1-1. C OPERATORS

Arithmetic Binary Operators	
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

Unary Operators	
*	Contents of Address
&	Address of Operand
-	Arithmetic Negate
!	Logical Negate
~	One's Complement
++	Increment
--	Decrement
(type)	Cast (Type Conversion)
sizeof	Size of Object (bytes)

Relational Binary Operators	
>	Greater Than
>=	Greater Than or Equal To
<	Less Than
<=	Less Than or Equal To
==	Equal To
!=	Not Equal To
&&	Logical AND
||	Logical OR

Assignment Operators	
=	Simple Assignment
+=	Add, then Assign
-=	Subtract, then Assign
*=	Multiply, then Assign
/=	Divide, then Assign
%=	Modulus, then Assign
<<=	Left Shift, then Assign
>>=	Right Shift, then Assign
&=	Bitwise AND, then Assign
^=	Bitwise Exclusive OR, then Assign
|=	Bitwise Inclusive OR, then Assign

Bitwise Binary Operators	
&	Bitwise AND
|	Bitwise OR (inclusive)
^	Bitwise OR (exclusive)
<<	Left Shift
>>	Right Shift

All functions in C are called by value, that is, the value of the parameter is passed to the function. In most languages, parameters are passed only by reference; the address of the parameter is passed to the function or procedure. However, since a pointer can be a parameter value, call by reference is also available in C.

Within functions, control flow statements specify the order in which computations are to be done. C has a number of structured control flow constructs, including if-else, else-if, while, do-while, for, and switch-case, as well as three varieties of jumps: continue, break, and goto.

STORAGE

The language C gives programmers explicit control over the way values are stored. Variables can be specified as local to the current invocation of a function, global to all functions in a separately compiled source file, or global to all functions in a program. When C programs execute, local variables are stored on a stack. When a local variable is known to be frequently used, the programmer can request that it be held in an AmZ8002 register; if the resources are available, the request will be honored.

ENVIRONMENT

A fundamental distinction is made between the system under which programs are developed (i.e. written, compiled, assembled, and linked) and the system under which these programs are executed. The former is called the host system, while the latter is called the target system. The host computer need not be the same type of computer as the target computer. A cross compiler is a compiler that is hosted on one type of computer and is targeted (i.e. generates code) for another type. ACC is a cross compiler for C that is hosted on the AmSYS8/8 development system and is targeted for the AmZ8002.

The term **environment** is used to refer to the combination of the hardware configuration and the operating system. Thus, the host environment of a cross compiler is the host computer along with its operating system. The compiler can access the resources of the host environment (in particular the compiler can use the host operating system), but since the code generated by the compiler will execute on the target system, this code can use only the resources of the target environment.

The host environment for ACC is defined by the AmSYS8/8 hardware, the AMDOS Operating System, and associated utility programs, including the MACRO8000 Assembler and the LINK8000 Linker. The target environment for ACC is defined by the AmZ8002 processor, the associated support hardware on the target board or boards, and the operating system (if any) available on the target system.

Certain functions in the Run-time Library (described later, see also Appendix D) make use of operating system resources; for example, **getchar** (a function that reads a character from the console) requires an I/O system. These functions are not available to C programs unless they are supported by the target environment. If the target environment does not include an operating system, or if the operating system is not integrated with the Run-time Library, these functions cannot be referenced at all in C programs (unless explicitly supported by user-written functions).

The minimal configuration for hosting the C compiler is an AmSYS8/8 with 64K bytes of memory, two floppy disks, and the AMDOS Operating System. The target processor for code generated by the C compiler is an AmZ8002 microcomputer. No operating system is required on the target system, but if the Am96/4016 Evaluation Board or Am96/4116 MonoBoard is used as a target, the ROM-based 96/4630 Monitor can be accessed from C programs.

Effectively, three different target environments can be defined for ACC:

- Stand-alone Environment. Any AmZ8002 system or board with no target operating system (for example, the 4116 board without the 4630 monitor).

- AMC supplied Monitor Environment. Any AmZ8002 system incorporating an AMC supplied target operating system. For example, the 4630 Monitor environment, which is either the 4016 Evaluation Board or a combination of the 4116 MonoBoard Computer and the 4630 Monitor.

- User supplied Monitor Environment. Any AmZ8002 system or board with a user-written operating system running on the target processor.

Figure 1-1 shows the host and target environments for ACC.

Later in this chapter, and also in Chapter 3, it will be shown that the choice of the target environment has implications for the functions that can be used in C programs and affects the procedures that are used to link and execute these programs.

THE AMC C SOFTWARE PACKAGE

The transformation of a C source program into an executable object module is accomplished by invoking in succession the C Compiler, the MACRO8000 Assembler, and the LINK8000 Linker.

The C compiler consists of three sequentially invoked components: the preprocessor, the parser, and the code generator; they each perform one pass of the compilation process. Each pass reads the output of the previous pass (or the source itself in the case of the first pass), makes the necessary transformation, and writes output into an intermediate file that can then be read by the next pass.

The output of the compiler's code generation pass is an AmZ8002 assembly language file that constitutes the input to the assembler, whose output is relocatable object code. This assembler output, along with several previously assembled library programs, in turn constitutes the input to the linker, which outputs executable code. This code can be downloaded into the target processor or PROM by using RTE16 software, PROM burning software, or the HOST program. Figure 1-2 shows a schematic overview of this process.

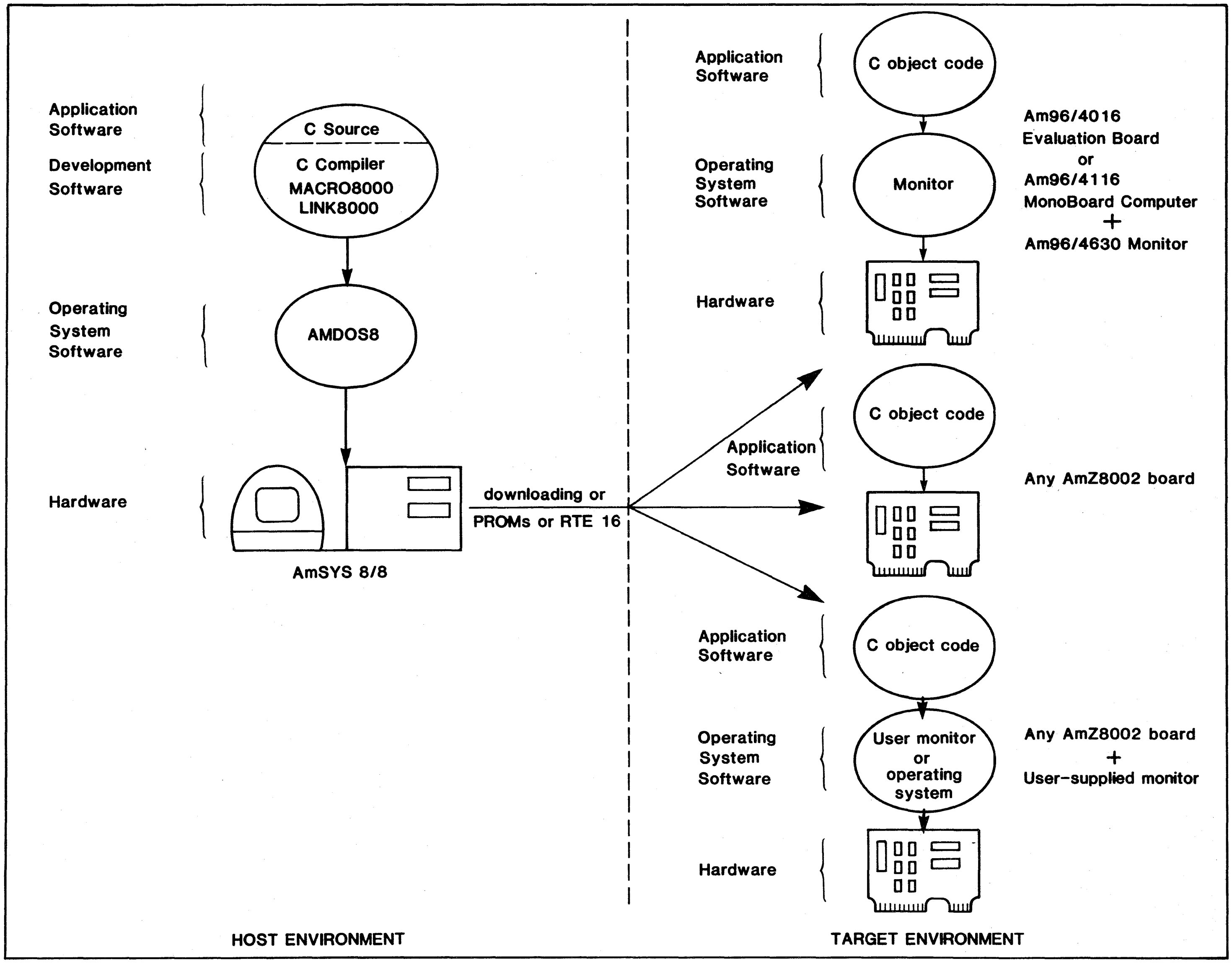


Figure 1-1. Compiler Environments

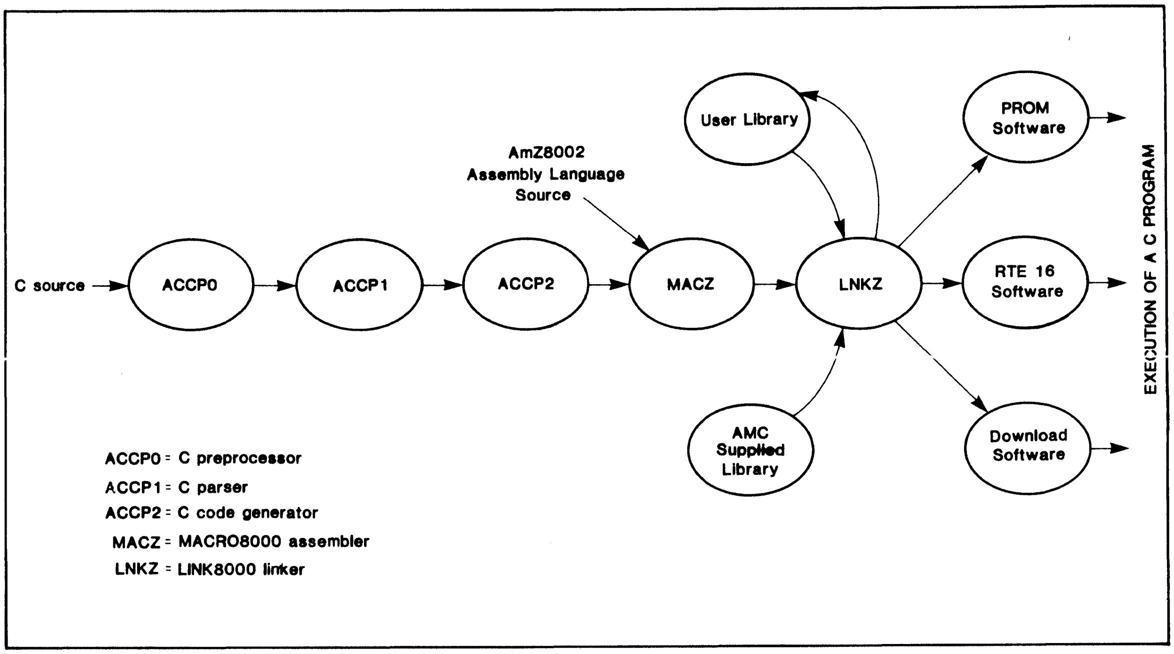


Figure 1-2. Compiling a C Program

The AMC C Software Package contains the programs required to carry out this process. They are described in the following paragraphs.

PREPROCESSOR (PASS 0)

The preproccesor performs a lexical analysis, that is, it converts the input source text into a sequence of tokens for use by the parser. Tokens are the smallest program units; they fall into four categories: identifiers, reserved words, special characters, and literal values. The preprocessor also supports conditional compilation, macro expansion, symbolic constants, and file inclusion.

Besides its use as pass 0 of the C compiler, the preprocessor can be used independently to produce, from multiple input streams, a single combined source file. For more details on the preprocessor, consult Appendix C.

PARSER (PASS 1)

Each programming language has its own set of grammatical rules which characterize the correct form of a program written in that language. The parser of the C compiler reads the token stream from the preprocessor and verifies that the source program satisfies the grammatical rules of C. The parser is said to perform a syntactical analysis. It uses the top-down method of syntax analysis known as recursive descent to parse everything except expressions, for which operator precedence is used.

The result of the syntax analysis is an intermediate code file, which is used as input by the code generator. Intermediate code consists of encoded representations of C language statements (parse trees) and of the flow of control in a function (flow graphs). See Appendix C for more information on the parser.

CODE GENERATOR (PASS 2)

The code generator reads the intermediate code from the parser and produces a .ZSC file containing assembly language code for MACRO8000.

The flow graph and the parse trees of the intermediate code are reconstructed into a flow and code graph in memory. All paths in the graph are traversed to eliminate jumps to jumps, jumps to code sequences that are otherwise unreachable, and other extraneous code. This process is repeatedly performed until no more improvement can be attained. Finally the assembly language code for the function is generated from the graph. Code generation is performed by an algorithm that looks through tables to find the correct code sequence for a given subexpression.

The code generator produces an assembly language module consisting of three segments, named CODE, LITS, and DATA. The first of these contains the actual instructions generated by pass 2; the second contains all literal values (character strings or string constants) appearing anywhere in the source program; the third segment contains space that is reserved for the values of external and static variables. See also Appendix C for more information on the code generator.

MACRO8000 ASSEMBLER

The assembler MACRO8000 reads the assembly language file from the code generator and generates a relocatable binary (.ZRL) file that can be targeted for the AmZ8002. MACRO8000 preserves the three-segment structure of the module generated by pass 2. See Appendix C and the MACRO8000 User's Manual for more information on the assembler.

RUN-TIME LIBRARY FILES

Input and output facilities are not part of the C language, instead I/O is performed by separate functions that are called by C programs. The ACC Run-time Library is a collection of functions designed to provide, among other things, a standard I/O system for C programs.

The functions in the Run-time Library can be separated into a standard set that is common to all target environments and a number of functions that are dependent in some way on the characteristics of the target. The functions in the standard set are collected together into one library file (CLIB). The machine dependent functions are grouped into several different library files (MLIBs), one for each target environment.

The file CLIB.ZRL on the distribution diskette is a LINK8000 library file containing a number of compiled and assembled C functions. These functions perform string manipulation, character and integer conversion, standard I/O, and several other useful operations. The functions in CLIB do not depend directly on the target environment (although four functions **gets, puts, scanf**, and **printf** , call functions in the machine dependent library). They have exactly the same effect as the equivalent functions in the Version 7 UNIX I/O library. (A complete list of the functions in CLIB is provided in Appendix D). A group of macros contained in the file ACCSTD.H are related to the CLIB functions. These macros perform certain tests on characters; they can be accessed from a C program if a **#include** statement is used at the beginning of the source file. The preprocessor performs the macro expansion.

The distribution diskette contains several target dependent library files, two of which are MLIB.ZRL and MLIB4630.ZRL. The first of these consists of functions that can be called in the stand-alone environment. The second contains functions that can be used in the 4630 Monitor environment.

MLIB.ZRL contains four functions (**inbyte**, **outbyte**, **inword**, and **outword**) that were written in assembly language and perform simple character or word I/O on a specified AmZ8002 I/O port. (See the AmZ8001/2 Processor Instruction Set Manual for a description of port I/O.) None of the functions in this set is comparable to any of the Version 7 UNIX functions; they are unique to the AmZ8002 environment. Two other assembly language functions (C@SAV and C@SWITCH) are also found in MLIB.ZRL. They are used by the compiler and cannot be called by user C programs.

Although it is not technically in a library file, the function C@INIT can be grouped with the functions in MLIB.ZRL. C@INIT is found in the file CINIT.ZRL; it is used by the compiler to start the user program and cannot be called by users directly. All the functions in MLIB.ZRL and CINIT.ZRL are described in Appendix D, where their code is listed.

MLIB4630.ZRL contains all the functions in MLIB.ZRL along with five others. Two C functions (**getchar** and **putchar**), which perform terminal I/O, and two assembly language functions (**moncall** and **exit**) can be called from user C programs. All these functions depend on the 4630 Monitor environment for their proper execution. The function **moncall** is used to call the 4630 Monitor, while **exit** is used to terminate user program execution and return to the Monitor. One other assembly language function is found in MLIB4630.ZRL: C@ABORT, which is used to process stack overflows. (See Chapter 2 for a discussion of stacks.)

C@ABORT cannot be called from user C programs. All the functions in MLIB4630.ZRL are described in Appendix D; their listings are also provided.

LINK8000 LINKER

The LINK8000 linker takes the relocatable .ZRL file produced by the assembler, combines it with the file CINIT.ZRL, with modules from the library file CLIB.ZRL and from some MLIB to produce an executable absolute (.BIN) output file. The three segments, CODE, DATA, and LITS, can be combined into a single memory-image, or else CODE and LITS can be separated from DATA for later burning into PROM.

The linker can also be used interactively to create a new target-dependent library file (MLIB) when the target environment is based on a user-supplied monitor (see Chapter 3). For more information on the linker, consult Appendix C and the LINK8000 User's Manual.

THE COMPILATION PROCESS

This section describes the commands that should be used in order to invoke the compilation process for C programs. The Stand-alone environment and AMDOS files written for this environment will be used to illustrate compiling for a typical target environment. Chapter 3 shows in detail the changes that are required to run in other environments.

In the discussion that follows, it is assumed that Drive A has a diskette containing the following files:

● AMDOS.SYS	The AMDOS system file
● SUBMIT.COM	An executable file containing the SUBMIT program
● MACZ.COM	MACRO8000
● LNKZ.COM	LINK8000
● ACCP0.COM	pass 0 of ACC
● ACCP1.COM	pass 1 of ACC
● ACCP2.COM	pass 2 of ACC
● CLIB.ZRL	The standard library file
● MLIB.ZRL	A target-dependent library file
● CINIT.ZRL	The function C@INIT
● ACCSTD.H	The standard header file
● ACC.SUB	An example submit file
● ACC.DIR	An example linker directive file

These files virtually fill a single-density diskette, so all source files should be saved on Drive B.

THE SUBMIT FILE ACC.SUB

Appendix C shows how each of the five programs described above is invoked. However, it can be tedious to sequentially type in each of these five command lines, so, ordinarily, users will make use of the AMDOS submit file facility. Consult the AMDOS User's Manual for a description of the use of the submit files.

The example submit file ACC.SUB can compile, assemble, and link a single C program. The contents of ACC.SUB are as follows:

```
ACCP0 -X -I ACC -O  $1.TMO $1.C
ACCP1 -C -O  $1.TM1  $1.TMO
ERA $1.TMO
ACCP2 -O $1.ZSC  $1.TM1
ERA $1.TM1
MACZ $1  O=$1,L=$1.PRN
ERA $1.ZSC
;DELETE NEXT TWO STATEMENTS FOR COMPILE - ONLY SUBMIT
LNKZ ACC B=$1, L=$1.LPR CZRL=$1
ERA $1.ZRL
```

Consulting Appendix C, we can interpret each of these lines. The filename B:SOURCE is substituted for the parameter $1. The final output file is B:SOURCE.BIN, with an assembler listing B:SOURCE.PRN and a linker load map B:SOURCE.LPR. Notice that the intermediate files B:SOURCE.TMO, B:SOURCE.TM1, and B:SOURCE.ZSC are erased from Drive B as soon as they have served their purpose. (The last entry in the LNKZ line is a symbolic constant override. Its function is described in Chapter 3.)

ACC.SUB can be invoked by typing the following command (it is assumed that the source file is SOURCE.C and that it is contained on Drive B):

```
A> SUBMIT  ACC B:SOURCE
```

(A> indicates the AMDOS prompt. The current drive is A.)

An abnormal termination occurs when an error is encountered during any of the passes of the compiler or during the assembly. The messages produced by the compiler when an error is encountered are documented in Appendix E. The submit file continues to process, even if errors are encountered. Complete termination can be forced by typing CONTROL-C after any pass. (Careful timing is required: the CONTROL-C must be entered between the end of one pass and the beginning of the next.)

This submit file assumes a single source file. If it is desired to link together separately compiled modules, the submit file can be split in two (where the comment indicates), with the first half used for compilation and assembly of user programs and the second half used for the final linking of separate source files.

NOTE

Very large source files (more than 300 lines) or files with a large number of symbols can cause the compiler to run out of memory (see Appendix E for error messages). This situation can be avoided by using shorter source files (200-300 lines) and **#include** files for global symbol definitions.

Nothing in the submit file commands reveals that the output code is targeted for the Stand-alone environment. All the environment-dependent information is stored in the linker directive file, which is referenced in the linker command line. The example directive file ACC.DIR is discussed in the following section.

THE DIRECTIVE FILE ACC.DIR

As described in the LINK8000 User's Manual, the linker can be operated either interactively or via a command file with the .DIR extension. The linker command line in the submit file shown earlier referenced the file ACC.DIR.

The directive file contains commands to the linker that:

1. Specify a PROGRAM (absolute) link,

2. Specify the .ZRL files to be linked,

3. Request that the library files CLIB and MLIB be entered to resolve externals,

4. Specify the order in which CODE, DATA, and LITS segments are to be linked,

5. Specify the absolute starting address for each group of segments, and

6. Specify the memory location of the stack(s). (See Chapter 2 for discussion of stacks.)

Proper use of the linker and the DIR file is the key to targeting ACC code toward different environments. Since it is difficult to interpret the DIR file without an understanding of stacks (explained in Chapter 2), further explanation of ACC.DIR is deferred until Chapter 3.

EXECUTION OF .BIN FILES

The output produced by the SUBMIT command (or LNKZ command) is an executable (.BIN) file. AMC provides several mechanisms for downloading this file to the AmZ8002 system for execution. The choice among these mechanisms depends on the environment for which the code is targeted.

4630 ENVIRONMENT

The AMDOS utility HOST can be used to download programs to boards supporting the 4630 Monitor. To invoke this utility, type

```
HOST
```

Then type the HOST command LDPR followed by the name of the .BIN file. For example:

```
LDPR B:SOURCE.BIN
```

This command loads SOURCE.BIN over a parallel link to the AmZ8002 board and transfers control to the Monitor.

NOTE
HOST does not use Multibus for downloading.

To begin execution of SOURCE, type the Monitor command

```
G
```

When the SOURCE program terminates, control returns to the Monitor. For more documentation on HOST and the Monitor, consult the Am96/4016 Evaluation Board User's Manual.

The AMDOS User's Manual also describes the PROM burning utilities PREHEAT, PPROG, and PLPROG. These programs can be used to reformat the .BIN file and download it over a serial link to a PROM-programming device (either a Data I/O or Pro-log PROM Programmer). Consult the AMC Application Note, Building PROM-Based C Systems, for more details on burning C programs into PROM.

STAND-ALONE ENVIRONMENT

The PROM-programming utilities can be used to burn PROMs for user boards as well as the AMC-supplied boards, but of course HOST is not be available for downloading in this environment.

Instead, the RTE16 support package can be used to load binary files down to their execution vehicle. In addition, RTE16 has several debugging facilities that enable programs to be tested. RTE16 software is described in the RTE16/8050 Real-Time Emulator Support Package User's Manual.

USER-SUPPLIED MONITOR ENVIRONMENT

The facilities for downloading and executing are identical in the Stand-Alone environment and in the User-Supplied monitor environment.

CHAPTER 2

COMPILER CODE GENERATION

INTRODUCTION

The C Compiler translates programs written in C into AmZ8002 assembly language. Consequently, the characteristic features of C programs -- data types, operators, control flow statements, functions, and storage classes -- are all mapped into the more restricted set of AmZ8002 operators and operands. When debugging a C program using RTE16, it is invariably necessary to look at selected sections of the assembler code that was produced, because the RTE16 debugging facilities operate at the assembly language level. In this chapter we will explore the code generation strategies of ACC.

REPRESENTATION OF DATA TYPES

C recognizes the following basic data types: bitfield, character (signed or unsigned), integer (short or long, signed or unsigned), floating point (standard precision or double precision). The AmZ8002 architecture provides operators that can manipulate bits, eight-bit bytes, words of two bytes, and long words of four bytes. The AMC C compiler provides a mapping between the data types of C and the data types of the AmZ8002. Table 2-1 illustrates this mapping.

REPRESENTATION OF OPERATORS

The C operator set was shown in table 1-1 of Chapter 1. Expressions involving these operators generally map into several AmZ8002 instructions because it usually takes several Loads to set up registers for the actual operation. By and large, however, for most C operators a single, unique AmZ8002 instruction performs the basic operation. The set of AmZ8002 operators is thus a fairly close match for the set of C operators.

Table 2-2 shows the same operators as table 1-1, but shown along with each one is the corresponding AmZ8002 operator that is produced by the compiler.

REPRESENTATION CONTROL FLOW STATEMENTS

The C language includes several structured control flow statements. The general code generation strategy for these statements is a test then a jump conditional. The differences in the statements result in different positions for the test and different targets for the jump. The chart in table 2-3 shows the skeleton code produced by the main control flow statements.

TABLE 2-1. DATA TYPES

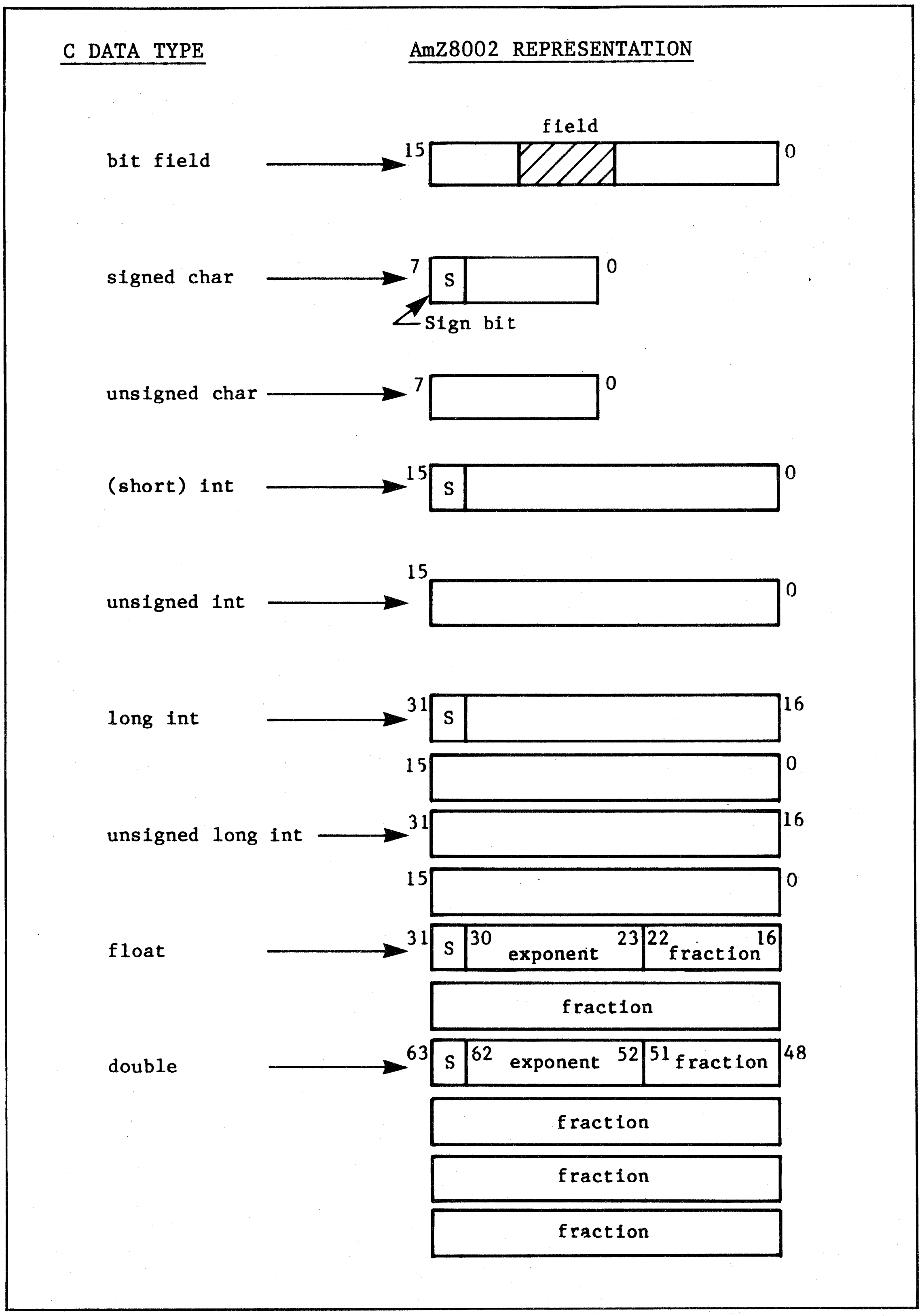


TABLE 2-2. C OPERATORS AND AmZ8002 OPERATORS

Arithmetic Binary Operators	
+	ADD, ADDB, ADDL
-	SUB, SUBB, SUBL
*	MULT, MULTL
/	DIV, DIVL
%	DIV, DIVL

Relational Binary Operators	
>	All these have the same
>=	general format:
<	
<=	CP expression
==	JR test
!=	where:
&&	test = GE, LT, LE, GT,
||	NZ, NE, Z, EQ, etc.

Bitwise Binary Operators	
&	AND
|	OR
^	XOR
<<	SLA, SDA, SDL
>>	SRA, SDA, SDL

Unary Operators	
*	LD Rn^
&	LD ^address
-	NEG
!	code sequence
~	COM
++	INC
--	DEC

Assignment Operators	
=	LD, LDB, LDL
+=	ADD, ADDB, ADDL
-=	SUB, SUBB, SUBL
*=	MULT, MULTL
/=	DIV, DIVL
%=	DIV, DIVL
<<=	SLA, SLL
>>=	SRA, SRL
&=	AND
^=	XOR
|=	OR

REPRESENTATION OF FUNCTIONS

To an assembly language or FORTRAN programmer, the run-time representation of functions in C can appear somewhat unusual. As functions are called, function variables are allocated memory, but when the function is exited, the allocated space is freed. If another function is called immediately thereafter, the same physical memory location might now hold the value of another variable used in the new function. To someone accustomed to looking in a fixed, absolute location for a variable's value, a dynamic language like C presents a very different view of memory.

FUNCTION CALLS AND RETURNS

Whenever a function is used in a C source program, for example in the statement

```
func (a, b, c);
```

the compiler produces a standard sequence of assembly language instructions, called the function calling sequence.

TABLE 2-3. CONTROL FLOW STATEMENTS

Control Flow Statement		Skeleton Code
if (expression)		CP expression
statement1;		JR test, LAB1
else		statement1
statement2;		JR LAB2
	LAB1:	statement2
	LAB2:	(next statement)
while (expression)	LAB1:	CP expression
statement		JR test, LAB2
		statement
		JR LAB1
	LAB2:	(next statement)
for (expression1; expression2;		expression1
expression3)	LAB1:	CP expression2
statement;		JR test, LAB2
		statement
		expression3
		JR LAB1
	LAB2:	(next statement)
do	LAB1:	statement1
statement1		CP expression
while		JR test, LAB1
expression		(next statement)
break		JR LAB
continue		JR LAB
goto		JR LAB
switch (c)		JP C@SWITCH
case ′char1′		
case ′char2′		(See Appendix D)
etc.		

test = GE, LT, LE, GT, NZ, NE, Z, EQ, etc.

The first few instructions in the function calling sequence get the values of each parameter (a, b, and c are parameters of the function func, in the above example) and store them on a dynamic memory structure called a stack. Figure 2-1 shows a diagram of the C run-time stack structure. Notice that there are actually two stacks, a downward-growing stack referenced by R15 and an upward-growing stack referenced by R12.

The upward-growing stack is called the frame stack and the downward-growing stack the return stack. Register R15 is called the stack pointer and R12 the frame pointer. Arguments are stored on the frame stack, as the diagram indicates. (Before the parameters are stored on the frame stack, the register variables, if any, of the calling function are pushed on this stack. Pushes move the frame pointer, whereas stores do not.) The return stack is used to hold the return address of the calling function and the previous frame pointer, as described in the next section.

The highest memory location in the return stack is called STACK@TOP, while the lowest location in the frame stack is called STACK@BTM. These two parameters set the size and location of the stack area of memory. These parameters are initialized by the linker DIR file (Chapter 3).

Parameters are passed by value, that is, a copy of the value of the parameter is stored on the stack, not the address of the parameter (except in the case of arrays). Passing parameters by value allows the called function to modify the parameters without changing the original values. Parameters of type char and float are expanded to int and double, respectively. Array names are converted to pointers if they appear as parameters.

Finally, after all the parameters have been stored on the frame stack, the function is called via the CALL instruction. Hence, an example of a function calling sequence might have the following form:

```
                LD   R7, ARG3:          %Get value of ARG3
Sample          LD   (6)^(R12), R7;     %Store it on the Stack
Calling         LD   R7, ARG2;          %Get value of ARG2
Sequence        LD   (4)^(R12), R7;     %Store it on the Stack
                LD   R7, ARG1;          %Get value of ARG1
                LD   (2)^(R12), R7;     %Store it on the Stack
                CALL func;              %Call the Function
```

(For simplicity, the addresses of the parameters are given as ARG1, ARG2, and ARG3 and each is assumed to be one word long. Actually, they would probably be referenced as a displacement from some location and their size could vary from one word to four words.) Notice that addresses on the frame stack are referenced as offsets from R12.

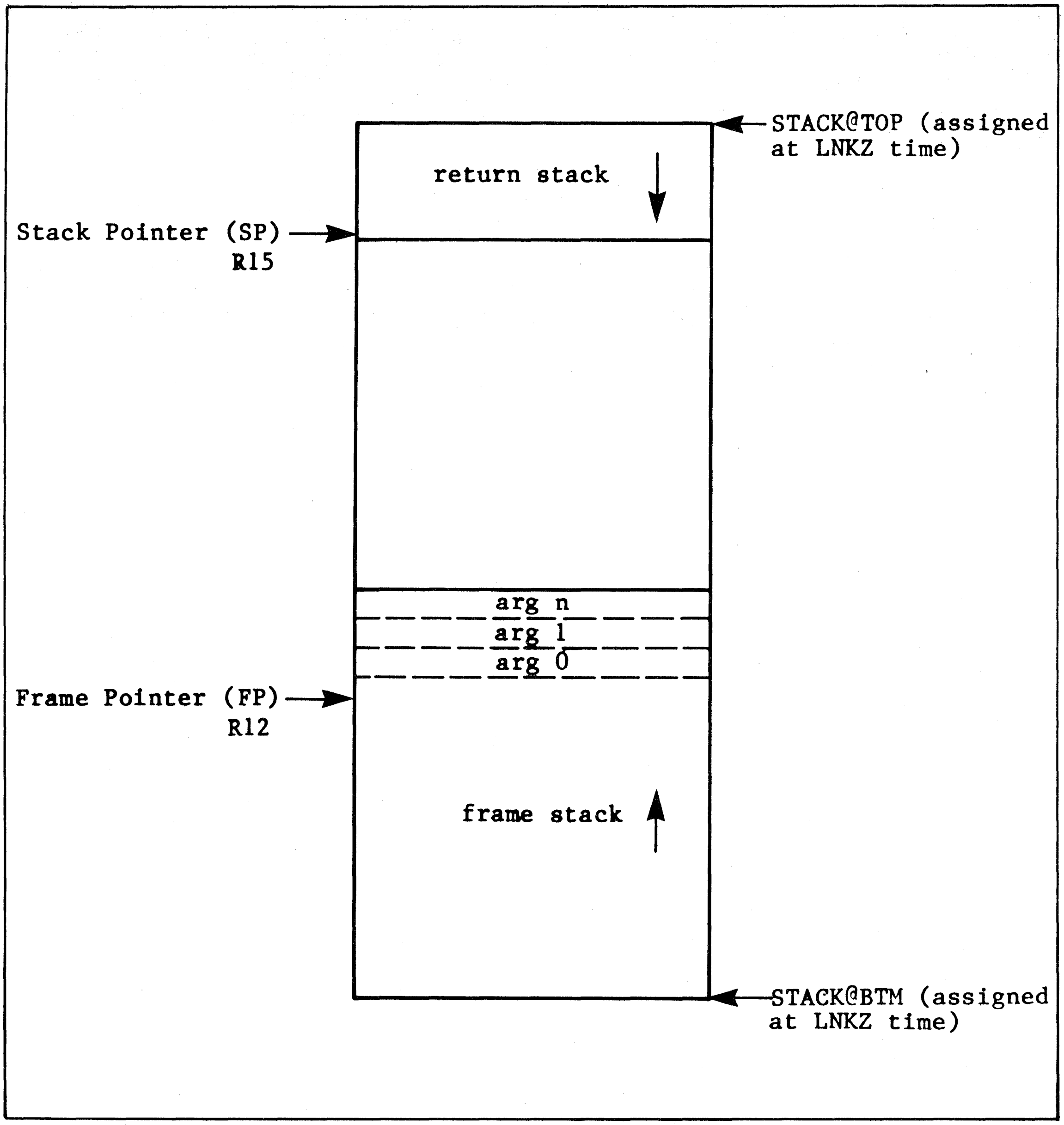


Figure 2-1. Stack Organization

STACK FORMAT

The fundamental unit of organization on the frame stack is the frame, consisting of a contiguous block of storage used to hold all the data needed by one function, including the parameters passed to it and all local variables. A frame is allocated from the frame stack at function invocation and released when the function returns to its caller.

The fundamental unit of organization on the return stack is the return block, consisting of storage for a pair of addresses: the return address of the calling function and the calling function's frame pointer.

The frame stack and the return stack are both maintained by the subroutine C@SAV, which is found in CLIB, the Run-time library. (See Appendix D.)

Figure 2-2 shows a sequence of diagrams that illustrate how the two stacks appear during each stage of a function call.

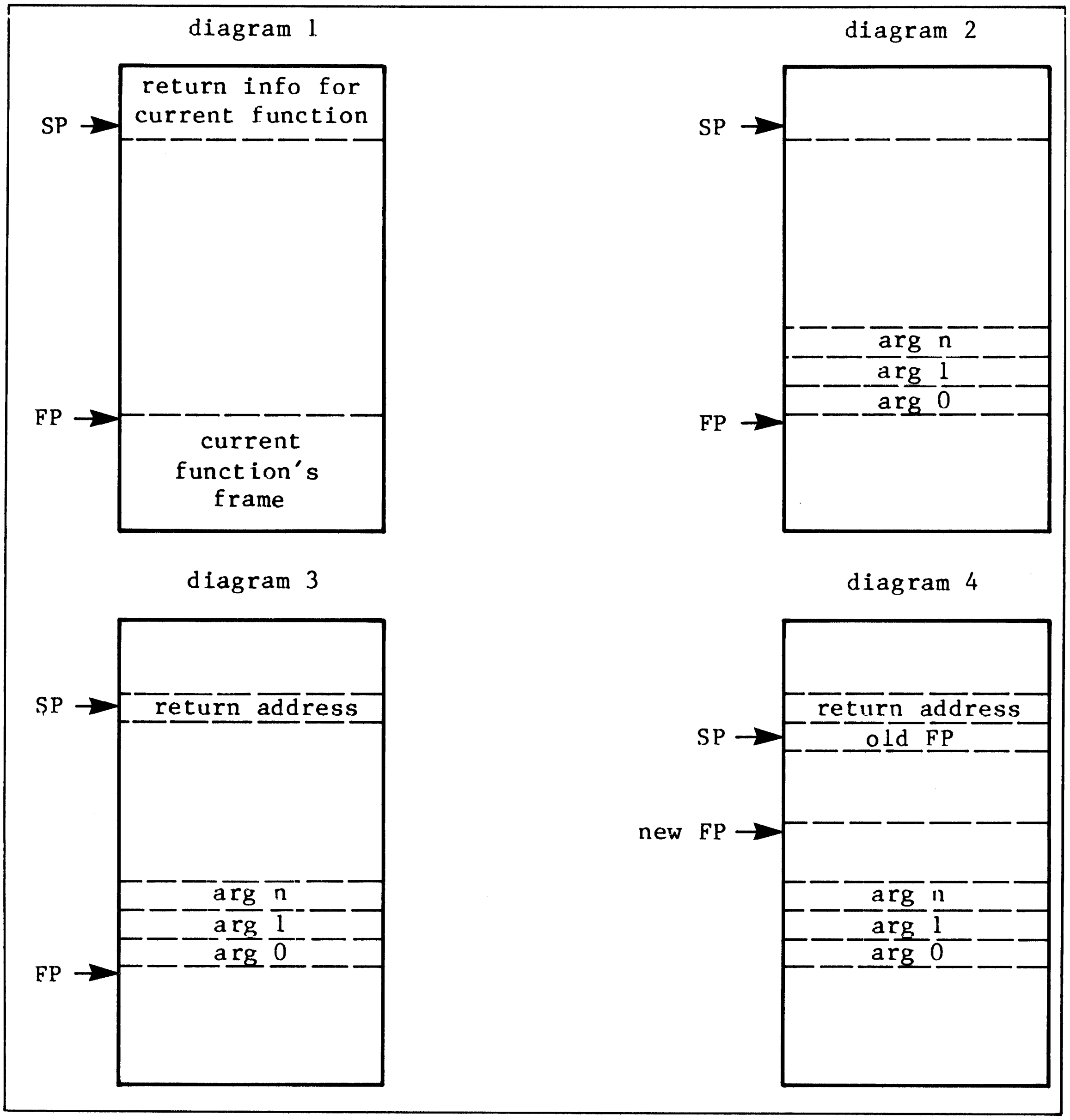


Figure 2-2. Stack Operation

Diagram 1: executing the original function

Initially the frame pointer points to the top location in the frame of the current function. The stack pointer points to the bottom location in the current function's return block (the old frame pointer).

Diagram 2: just before the CALL to the new function

Parameters are stored on the stack before the function is called (see the previous section). They appear just above the frame pointer of the calling function.

Diagram 3: immediately after the CALL, but before C@SAV is called

The CALL instruction has the effect of pushing the return address of the calling function on the return stack.

Diagram 4: after C@SAV is called

The first two assembly language instructions of any C function are:

```
LD        R0, framesize;
CALL      C@SAV;
```

The function C@SAV then has the responsibility of allocating a frame for the called function by adding to the frame pointer a number equal to the size of the new frame. The old frame pointer value is then pushed on the return stack. (The actual assembly language code for C@SAV is shown in Appendix D.)

Automatic variables (see the section entitled Representation of Storage Classes) are stored immediately above the parameters in the same order that they appear in the variables list of the called function. Any temporaries generated by the compiler follow the automatic variables.

Functions written in assembly language which are to be interfaced with C programs (see Chapter 4 for an example) need not call C@SAV. Any arguments passed to that function can be altered by the function, but no changes can be made any lower than the location of argument 0.

REGISTERS

The following AmZ8002 registers are used by functions or to support function calls.

- Register R15 contains a downward growing stack pointer that always points to the last item pushed onto the return stack.

- Register R12 contains the upward growing current frame pointer; it always points to the end of the current frame; that is, to the highest location in the current frame. Most references to items on the frame stack are by displacements from the current frame pointer.

- Registers R8, R9, and R10 are allocatable; that is, they can hold variables of the storage class register.

- Register R7 contains the result of the function call (when it is no more than a word in length). Long integer results are returned in the register pair RR6; float and double values are returned in RQ4.

- Registers R0-R7 are used for expression evaluation and for temporary storage.

REPRESENTATION OF STORAGE CLASSES

The C concept of the storage class really involves two different but related notions: the scope of a variable, and where that variable is stored. Variables can have three possible scopes:

- Available to all functions in the program (global scope).
- Available to all functions in the module currently being compiled (modular scope).
- Available only within the defining function (local scope).

There are three places where variables can be stored:

- On the stack, allocated dynamically (automatic storage).
- In the DATA segment produced by the compiler (static or external storage, depending on the scope).
- In an AmZ8002 register (register storage).

Combining scope and storage location yields the four different storage classes for variables:

- register variables are stored in registers, their scope is limited to the current function (i.e. they are local variables).

- automatic variables are stored on the frame stack, their scope is also local.

- external variables are stored in the DATA segment, their scope is global.

- static variables are stored in the DATA segment; however, their scope depends on whether they were declared inside or outside of a function. If inside, their scope is local; if outside, their scope is modular.

ALLOCATING STORAGE

The ACC strategy for allocating storage for variables is fairly simple:

- Externals. Storage is allocated in the DATA segment corresponding to the module where the variable is initialized.

- Static variables (both kinds). Storage is allocated exactly as with externals, but these variables are not declared external to the linker.

- Automatic. Storage is allocated on the frame stack by C@SAV when the function is called. Storage is deleted at the return.

- Register. Storage is allocated in R8, R9, and R10 as they become available.

ADDRESSING

The ACC strategy for addressing variables is also straight-forward.

- Externals and static variables are both addressed using direct addressing. However, the assembly language code addressing externals can reference the user-defined name of the variable, while statics are referenced using a unique (but sometimes cryptic) compiler-generated label.

- Automatic variables are addressed via the indexed addressing mode, using R12 as the base.

- Register variables are accessed through the register addressing mode.

CHAPTER 3
TARGET DEPENDENCIES

INTRODUCTION

In Chapter 1, three classes of target environments were defined:

- Stand-alone (no monitor) environment
- AMC supplied monitor (e.g. 4630) environment
- User supplied monitor environment

At the end of Chapter 1, an example submit file was used to compile, assemble, and link a C program targeted for the Stand-alone environment. The actual target-dependent information used by the submit file was contained in a linker directive file. The purpose of this chapter is to examine that directive file in detail and to show how .DIR files and other files must be modified to produce code targeted for other environments. Consult the LINK8000 User's Manual for more information on linking.

THE DIRECTIVE FILE ACC.DIR

The following listing includes a line-by-line commentary on the directive file ACC.DIR that was referenced in the submit file example ACC.SUB of Chapter 1. The target environment specified is the Am96/4116 MonoBoard Computer without 4630 Monitor (i.e., a Stand-alone environment).

```
PROGRAM C@INIT;
```

This is an absolute link; the starting address of the program is the label C@INIT.

```
CONST CZRL = CTMP;
```

Set CZRL to a dummy value, CTMP, which is overridden by the parameter $1 from the linker command line.

```
FILE CINIT,
     CZRL;
```

Link files CINIT and CZRL (which evaluates to $1) in that order, so the function in CINIT will be the lowest in memory.

```
SEARCH   CLIB;
SEARCH   MLIB;
```

Search the target-independent library (CLIB) and the target-dependent library (MLIB) for any unresolved external values.

```
ABSOLUTE #8100;               Set the first memory location to
COMBINE BY SEGMENT;           #8100.  Link all segments with the
                              same name; then link the groups of
                              segments in alphabetical order.

ASSIGN STACK@BTM :=$^;        Set the stack bottom to the current
                              instruction pointer value.  (i.e. the
                              highest location reached after all
                              the segments have been linked.)

ASSIGN STACK@TOP := #FE00^;   Set the stack top to location #FE00.

MAP;                          Display the link map.

END.                          End the link.
```

The target dependencies specified by the directives in this file can be grouped into three classes:

- memory organization
- library functions supported
- initialization

MEMORY ORGANIZATION

The directives beginning with ABSOLUTE #8100 and ending with ASSIGN STACK@TOP = #FE00^ are concerned with specifying the memory organization in the target system. Consult the Am96/4116 MonoBoard Computer User's Manual for a description of the memory layout of the 4116. Addresses 0000 through 7FFF are reserved for ROM. Addresses 8000 through 80FF are RAM space required by the Monitor (when it is present). Thus 8100 is the first available RAM address. The highest memory location on the 4116 is FE00, which has been assigned to the top stack. This directive file, therefore, locates all segments and the stack into RAM. RAM is organized as follows (from highest location to lowest):

```
        ______________ FE00
       |   Stack      |
       |   LITS       |
       |   DATA       |
       |___CODE_______| 8100
       |              |
       |              |
       |   (ROM)      |
       |              |
       |______________| 0000
```

USING THE 4016 EVALUATION BOARD

When the 4016 Evaluation Board is the target environment, different parameters must be used in the ABSOLUTE and ASSIGN STACK@TOP directives. Edit the DIR file so that these directives read:

```
ABSOLUTE #4100;
ASSIGN STACK@TOP :=#5FFE^;
```

The directive file now generates a memory image with the following format:

```
 ____________   5FFE
|   Stack    |
|   LITS     |
|   DATA     |
|___CODE_____|  4100
|            |
|            |
|   (ROM)    |
|            |
|____________|  0000
```

ROM AND RAM SYSTEMS

No matter what monitor (if any) exists on the target system, when PROMs are to be burned, the directives must to be changed. For example, the following directive file fragment illustrates the directives that should be used to locate the CODE and LITS segments in ROM:

```
ABSOLUTE starting address in ROM;
COMBINE .CODE;
COMBINE .LITS;

ABSOLUTE starting address in RAM;
COMBINE .DATA;
ASSIGN STACK@BTM := $^;
ASSIGN STACK@TOP := highest RAM address;
```

(For more information on ROM/RAM systems, see the AMC Application Note, Building PROM-Based C Systems.)

LIBRARIES

SEARCH directives in DIR files specify the libraries to be entered and searched in order to resolve externals. As stated in Chapter 1, the library functions can be classified into a standard set of functions that are common to all target environments (CLIB), and a number of different MLIBs, each containing those functions that are specific to a particular environment.

The SEARCH directives in ACC.DIR reference CLIB and MLIB, the target-dependent library specific to the Stand-alone environment. MLIB4630 could be substituted for MLIB if the program called the functions **getchar**, **putchar**, **moncall**, **exit**, or C@ABORT. The code for these functions is shown in Appendix D. Notice that **moncall**, **exit**, and C@ABORT are assembly language functions that all result in an SC (System Call) instruction. On the other hand, **getchar** and **putchar** are C functions that simply call **moncall**.

The four CLIB functions **scanf**, **printf**, **puts**, and **gets** all call the functions **putchar** and **getchar** to do low-level I/O. So, for MLIB to be used as the target library, no calls can be made to **printf**, **scanf**, **puts**, and **gets**, as well as to the five MLIB4630 functions described above.

In order for programs that call any of these four functions to operate successfully, they must be linked with a target-dependent library that includes **putchar** and **getchar**. In lieu of a 4630 monitor, **getchar** and **putchar** must be interfaced to a user-supplied monitor. The same requirement holds for **exit** (if it is referenced) and C@ABORT (called by C@SAV in case of stack overflow). The specifications for **putchar**, **getchar**, **exit**, and C@ABORT replacements can be determined from the code shown in Appendix D.

INITIALIZATION

After a C program has been compiled, assembled, linked, and down-loaded to an execution board, the last step is to execute the code. The initialization issue arises here. Before the code can be executed, a certain amount of preliminary housekeeping work must be done. The stack pointer and frame pointer must be loaded with their correct starting values (STACK@TOP and STACK@BTM, respectively) and the Program Counter must be loaded with the starting address of **main**. The function that performs this initialization needs to know details about the target environment, such as where the stack is to be placed in memory and where execution normally begins. Consequently, the initialization function is very target dependent.

4630 MONITOR ENVIRONMENT

In the 4630 environment, the function that performs this initialization is called C@INIT; it is specified as the program entry point in the PROGRAM directive of the file ACC.DIR, and its .ZRL file is also specified in the FILE directive. The Monitor itself calls C@INIT, which sets up the stack and calls **main**. The assembly language code of 4630 C@INIT is shown in Appendix D. Notice that the values of STACK@TOP and STACK@BTM are external (They are supplied at link time by the DIR file). Notice also that **main** is called, which means that when the user program completes, control will return to C@INIT. The last instruction in C@INIT will return to the Monitor. (SC with non-zero parameter is interpreted by the 4630 Monitor as a return.)

This C@INIT works for RAM-based 4630 systems, but it is not sufficient for most ROM-based systems. In ROM code, external variables must be initialized at execution time instead of compile time, so C@INIT must perform this initialization. The AMC Application Note, Building PROM-based C Systems, describes how to write a modified C@INIT that can initialize external variables.

STAND-ALONE ENVIRONMENT

In the stand-alone environment, C@INIT has a number of additional responsibilities beyond those described above. Besides handling the initialization of the C program, C@INIT has to do all the hardware initialization jobs that are normally performed by a monitor. For example, setting the memory refresh counter, if the CPU is used to refresh RAM; handling the power start-up sequence; setting system or normal mode; initializing I/O devices; and anything else that the board requires.

USER-SUPPLIED MONITOR ENVIRONMENT

The tasks performed by C@INIT in the 4630 environment will have to be done by some function in the user-monitor environment; so will the additional tasks performed by C@INIT in the stand-alone environment. But the division of labor between the monitor and C@INIT will be entirely dependent on the user′s system.

CHAPTER 4

AN ANNOTATED EXAMPLE

The following section is organized around five listings: the C source listings of a bubblesort program and an exchange function, the AmZ8000 assembly language listing produced by using a submit file, the source code for an assembly-language exchange function that will be substituted for the equivalent C function, and the linker load map for the new optimized sort. From these annotated listings, it is easy to understand the run-time environment of this program; thus, debugging at the assembly language level should be straight-forward.

THE C SOURCE PROGRAM

The source code shown in figure 4-1 implements a bubblesort routine. Notice that the function **exch (&a,&b)** is called in the innermost loop to exchange items a and b. This function is not shown in the source code; it is external.

```
/* This program reads a line from the console, sorts it, and */
/* prints out the sorted line on the console.                */


#define BUFSIZE 80

main() {
        char buf[BUFSIZE+1];
        int  i,j,n;

        gets(buf);
        n = strlen(buf) - 1;
        for (i=1;i<=n;++i){
                for (j=n;j>=i;--j){
                        if (buf[j-1]>buf[j])
                         exch(&buf[j], &buf[j-1]);
                }
        }
        puts(buf);
}
```

Figure 4-1. Bubblesort Function

A C exch FUNCTION

Figure 4-2 shows the source code for one implementation of **exch**. This implementation is written in C. Notice that the parameters passed to **exch** are pointers to the items being swapped, not the items themselves. Pointers are used because of the call-by-value convention of C functions. Remember that when parameters are passed, only copies of the actual values are sent; so if the called function modified the parameters, the original values will not be affected. Thus, if the actual items to be swapped were passed, the array buf would not be modified at all.

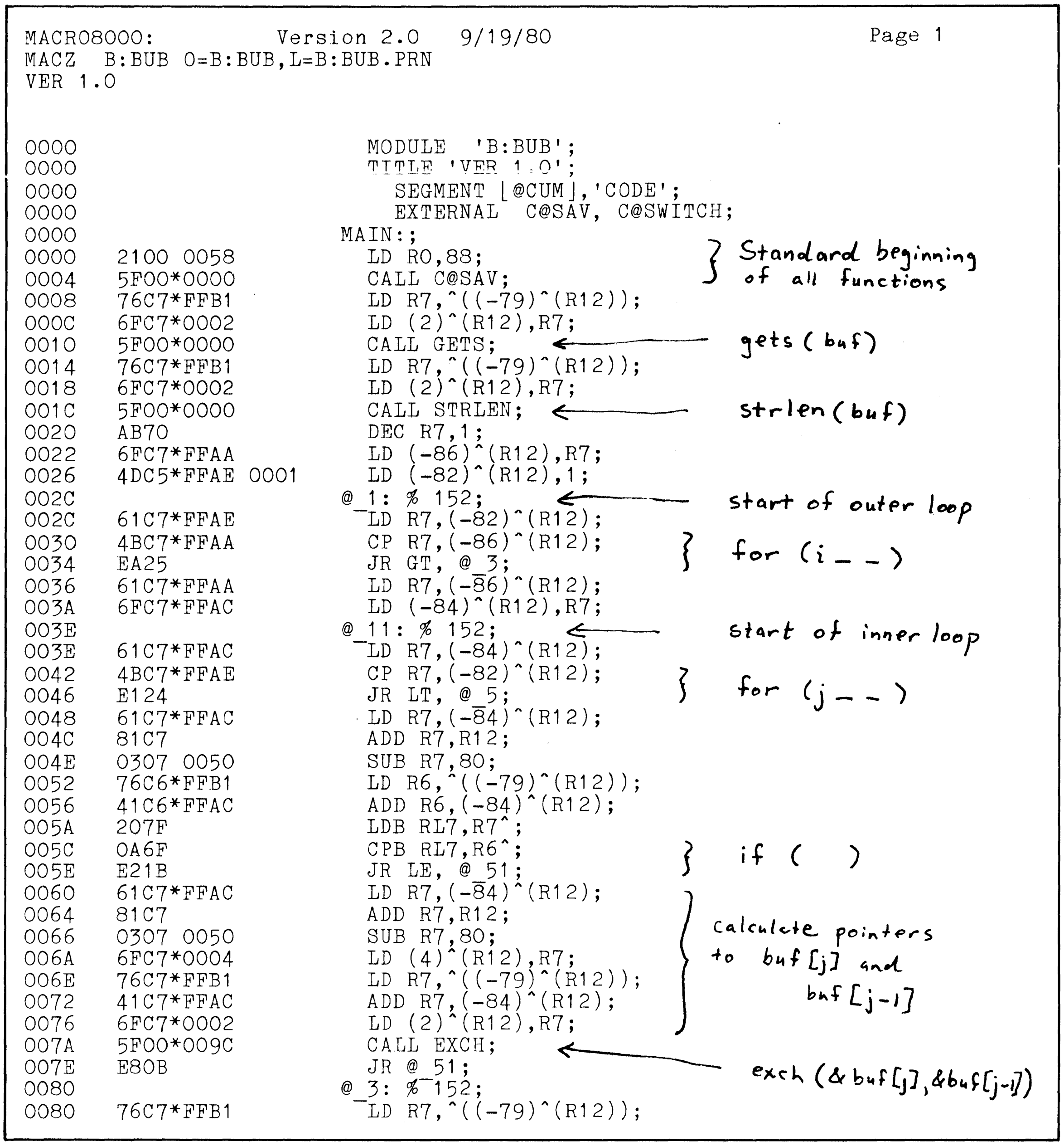
```
/* This function interchanges the two characters pointed to */
/* by ps and pt                                             */




exch(ps, pt)

register char *ps, *pt;

{
     char temp;

     temp = *ps;
     *ps = *pt;
     *pt = temp;
}
```

Figure 4-2. C **exch** Function

OUTPUT FROM A SUBMIT

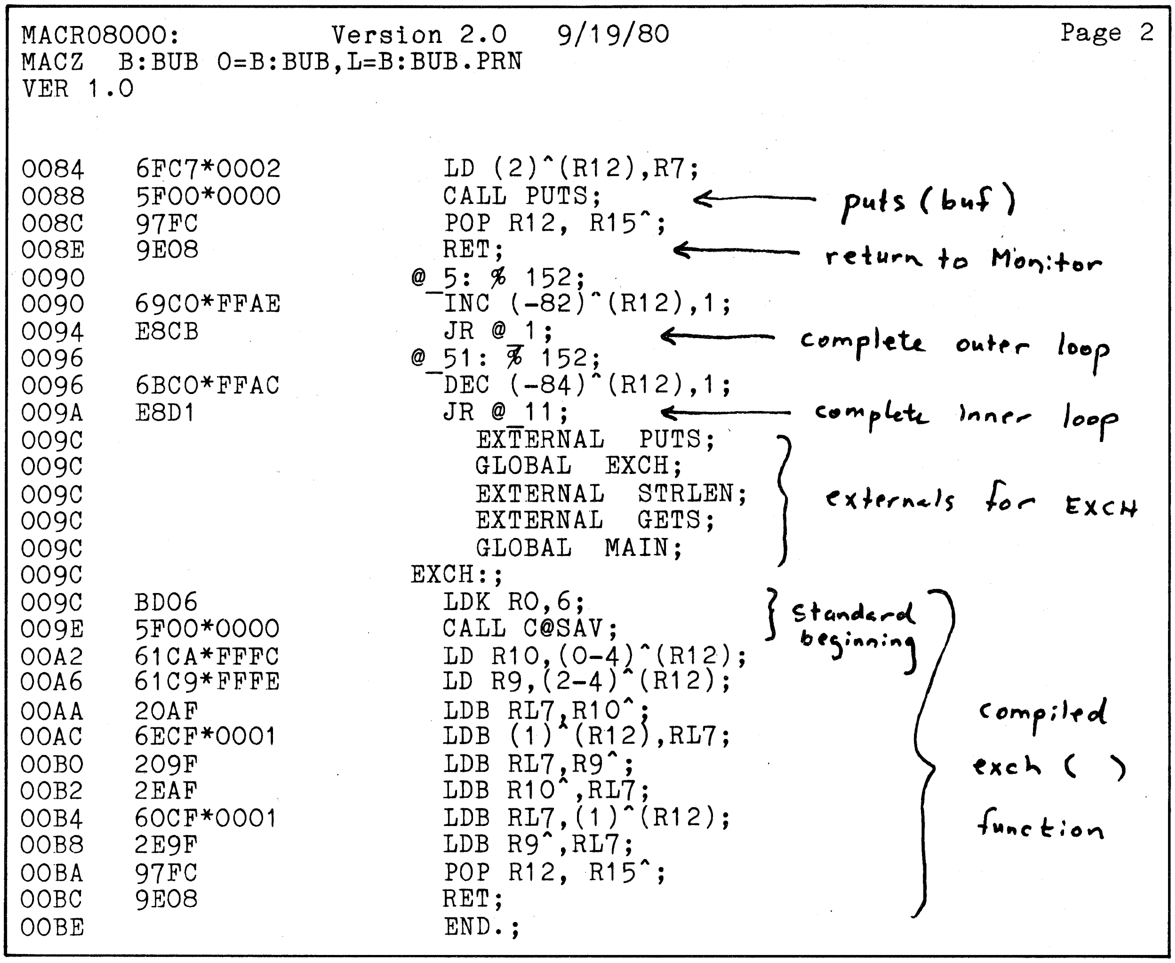
Figure 4-3 shows the .PRN file produced when **main** and **exch (&a,&b)** are compiled in one module, using the submit file explained in Chapters 1 and 3.

The marginal annotations identify C control flow statements and operators that were translated into AmZ8002 code.



```
MACRO8000:          Version 2.0   9/19/80                        Page 1
MACZ  B:BUB O=B:BUB,L=B:BUB.PRN
VER 1.0


0000                          MODULE  'B:BUB';
0000                          TITLE 'VER 1.0';
0000                            SEGMENT ⌊@CUM⌋,'CODE';
0000                            EXTERNAL  C@SAV, C@SWITCH;
0000                        MAIN:;
0000     2100 0058            LD R0,88;                    } Standard beginning
0004     5F00*0000            CALL C@SAV;                  } of all functions
0008     76C7*FFB1            LD R7,^((-79)^(R12));
000C     6FC7*0002            LD (2)^(R12),R7;
0010     5F00*0000            CALL GETS;  ←————————————        gets ( buf)
0014     76C7*FFB1            LD R7,^((-79)^(R12));
0018     6FC7*0002            LD (2)^(R12),R7;
001C     5F00*0000            CALL STRLEN;  ←——————————        strlen(buf)
0020     AB70                 DEC R7,1;
0022     6FC7*FFAA            LD (-86)^(R12),R7;
0026     4DC5*FFAE 0001       LD (-82)^(R12),1;
002C                        @_1: % 152;  ←—————————————        start of outer loop
002C     61C7*FFAE            LD R7,(-82)^(R12);
0030     4BC7*FFAA            CP R7,(-86)^(R12);         }     for (i _ _ )
0034     EA25                 JR GT, @_3;                }
0036     61C7*FFAA            LD R7,(-86)^(R12);
003A     6FC7*FFAC            LD (-84)^(R12),R7;
003E                        @_11: % 152;  ←————————————        start of inner loop
003E     61C7*FFAC            LD R7,(-84)^(R12);
0042     4BC7*FFAE            CP R7,(-82)^(R12);         }     for (j _ _ )
0046     E124                 JR LT, @_5;                }
0048     61C7*FFAC            LD R7,(-84)^(R12);
004C     81C7                 ADD R7,R12;
004E     0307 0050            SUB R7,80;
0052     76C6*FFB1            LD R6,^((-79)^(R12));
0056     41C6*FFAC            ADD R6,(-84)^(R12);
005A     207F                 LDB RL7,R7^;
005C     0A6F                 CPB RL7,R6^;               }     if (    )
005E     E21B                 JR LE, @_51;               }
0060     61C7*FFAC            LD R7,(-84)^(R12);         }
0064     81C7                 ADD R7,R12;                }
0066     0307 0050            SUB R7,80;                 }   calculate pointers
006A     6FC7*0004            LD (4)^(R12),R7;           }   to  buf [j] and
006E     76C7*FFB1            LD R7,^((-79)^(R12));      }          buf [j-1]
0072     41C7*FFAC            ADD R7,(-84)^(R12);        }
0076     6FC7*0002            LD (2)^(R12),R7;           }
007A     5F00*009C            CALL EXCH;  ←————————————
007E     E80B                 JR @_51;                       exch (&buf[j],&buf[j-1])
0080                        @_3: % 152;
0080     76C7*FFB1            LD R7,^((-79)^(R12));
```

Figure 4-3. Compiler Output



```
MACRO8000:          Version 2.0   9/19/80                    Page 2
MACZ  B:BUB O=B:BUB,L=B:BUB.PRN
VER 1.0


0084   6FC7*0002           LD (2)^(R12),R7;
0088   5F00*0000           CALL PUTS;
008C   97FC                POP R12, R15^;
008E   9E08                RET;
0090                     @_5: % 152;
0090   69C0*FFAE           INC (-82)^(R12),1;
0094   E8CB                JR @_1;
0096                     @_51: % 152;
0096   6BC0*FFAC           DEC (-84)^(R12),1;
009A   E8D1                JR @_11;
009C                         EXTERNAL  PUTS;
009C                         GLOBAL  EXCH;
009C                         EXTERNAL  STRLEN;
009C                         EXTERNAL  GETS;
009C                         GLOBAL  MAIN;
009C                     EXCH:;
009C   BD06                LDK R0,6;
009E   5F00*0000           CALL C@SAV;
00A2   61CA*FFFC           LD R10,(0-4)^(R12);
00A6   61C9*FFFE           LD R9,(2-4)^(R12);
00AA   20AF                LDB RL7,R10^;
00AC   6ECF*0001           LDB (1)^(R12),RL7;
00B0   209F                LDB RL7,R9^;
00B2   2EAF                LDB R10^,RL7;
00B4   60CF*0001           LDB RL7,(1)^(R12);
00B8   2E9F                LDB R9^,RL7;
00BA   97FC                POP R12, R15^;
00BC   9E08                RET;
00BE                       END.;
```

Figure 4-3. Compiler Output (continued)

AN ASSEMBLY LANGUAGE exch FUNCTION

Figure 4-4 shows an assembly language version of **exch** that has been hand-coded. Notice that a few bytes have been saved by not calling C@SAV. (Since **exch** makes no function calls, it is not necessary for it to call C@SAV.)

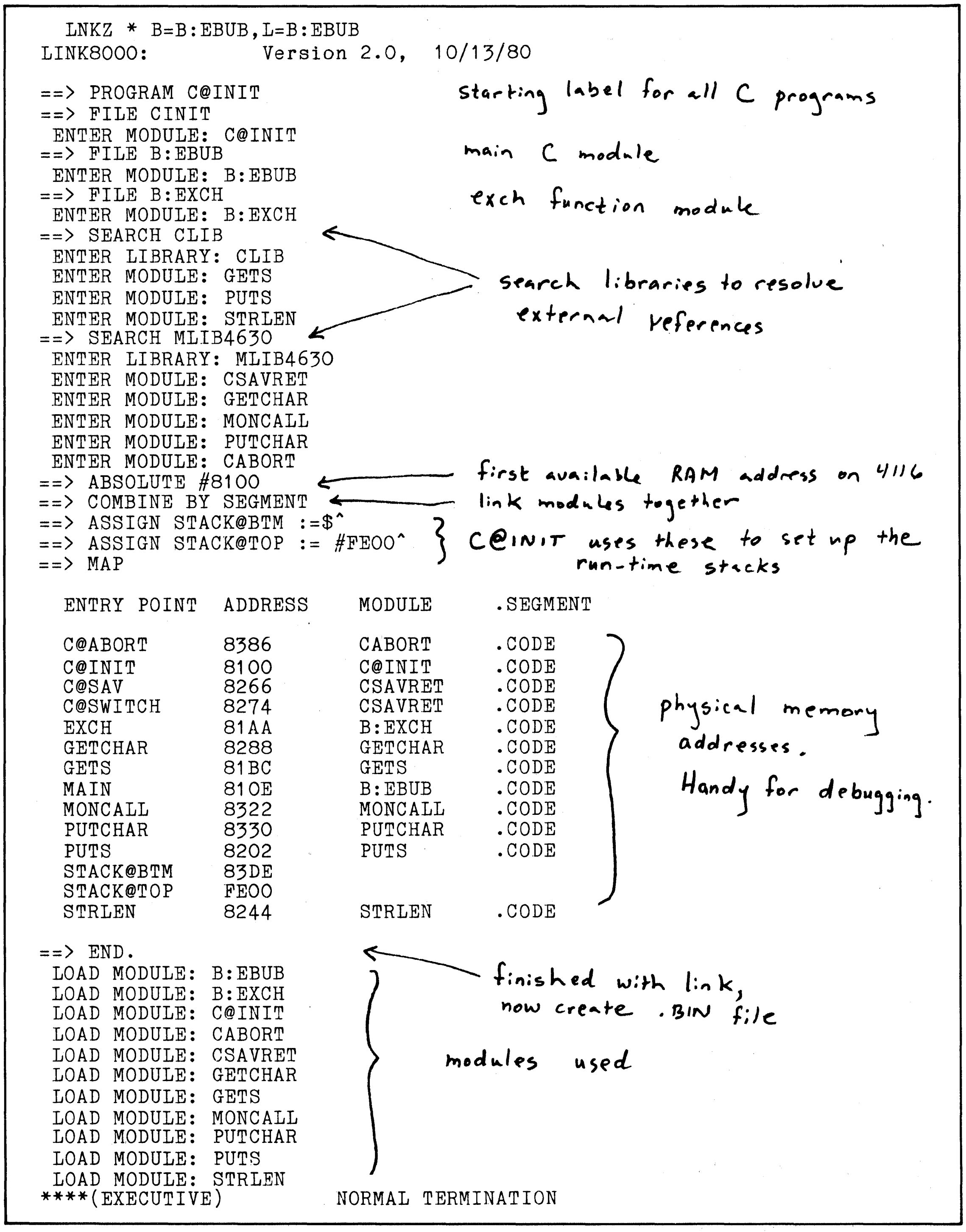
The marginal annotations identify the features necessary for interfacing this assembly language function with a C program.

```
          MODULE 'B:EXCH';

% The function exch interchanges two elements
% in a character array.
%
% Exch can be called from C as follows:
%
%
%         exch(ps,pt)
%
%
% where ps and pt are pointers to the two
% elements of a character array that are to
% be swapped.
%
%
%
% This Z8002 assembly language module
% implements the C callable exch function.
% The two pointer arguments are located at
% offsets 2 and 4 from the frame pointer R12.
% No value is returned by this function.

          TITLE      'EXCH';
          GLOBAL     EXCH;        <---- make EXCH available to
          SEGMENT    'CODE';              other functions

EXCH:     LD         R5, (2)^(R12);  <- move address of character to R5
          LDB        RH6, R5^;       <- RH6 gets first character
          LD         R4, (4)^(R12);  <- move address of other char to R4
          LDB        RL6, R4^;       <- RL6 gets other character
          LDB        R4^, RH6;       } swap characters indirect through
          LDB        R5^, RL6;       }        R4 and R5
          RET;
          END.                      <-- return to calling function
```

Figure 4-4. Assembly Language **exch** Function

THE LINKER LOAD MAP

Figure 4-5 shows a linker load map for the interactive link used to build a .BIN file from the file containing **main** and the file containing the assembly language **exch.** The bubblesort itself was recompiled first, using a compile-only submit file. Then EXCH.ZSC was assembled separately and linked with the new bubblesort (EBUB.ZRL).



```
  LNKZ * B=B:EBUB,L=B:EBUB
LINK8000:          Version 2.0,   10/13/80

==> PROGRAM C@INIT                          starting label for all C programs
==> FILE CINIT
 ENTER MODULE: C@INIT
==> FILE B:EBUB                             main C module
 ENTER MODULE: B:EBUB
==> FILE B:EXCH                             exch function module
 ENTER MODULE: B:EXCH
==> SEARCH CLIB
 ENTER LIBRARY: CLIB
 ENTER MODULE: GETS                         search libraries to resolve
 ENTER MODULE: PUTS                           external references
 ENTER MODULE: STRLEN
==> SEARCH MLIB4630
 ENTER LIBRARY: MLIB4630
 ENTER MODULE: CSAVRET
 ENTER MODULE: GETCHAR
 ENTER MODULE: MONCALL
 ENTER MODULE: PUTCHAR
 ENTER MODULE: CABORT
==> ABSOLUTE #8100                          first available RAM address on 4116
==> COMBINE BY SEGMENT                      link modules together
==> ASSIGN STACK@BTM :=$^
==> ASSIGN STACK@TOP := #FE00^             C@INIT uses these to set up the
==> MAP                                              run-time stacks

  ENTRY POINT  ADDRESS    MODULE     .SEGMENT

  C@ABORT      8386       CABORT     .CODE
  C@INIT       8100       C@INIT     .CODE
  C@SAV        8266       CSAVRET    .CODE
  C@SWITCH     8274       CSAVRET    .CODE          physical memory
  EXCH         81AA       B:EXCH     .CODE            addresses.
  GETCHAR      8288       GETCHAR    .CODE
  GETS         81BC       GETS       .CODE            Handy for debugging.
  MAIN         810E       B:EBUB     .CODE
  MONCALL      8322       MONCALL    .CODE
  PUTCHAR      8330       PUTCHAR    .CODE
  PUTS         8202       PUTS       .CODE
  STACK@BTM    83DE
  STACK@TOP    FE00
  STRLEN       8244       STRLEN     .CODE

==> END.
 LOAD MODULE: B:EBUB                     finished with link,
 LOAD MODULE: B:EXCH                     now create .BIN file
 LOAD MODULE: C@INIT
 LOAD MODULE: CABORT
 LOAD MODULE: CSAVRET                  modules used
 LOAD MODULE: GETCHAR
 LOAD MODULE: GETS
 LOAD MODULE: MONCALL
 LOAD MODULE: PUTCHAR
 LOAD MODULE: PUTS
 LOAD MODULE: STRLEN
****(EXECUTIVE)          NORMAL TERMINATION
```

Figure 4-5. Linker Load Map

EXECUTION

This example used the system dependent functions **gets** and **puts** , so it must be executed under a target operating system. In the 4630 environment, HOST can be used to download BUB.BIN to the Monitor. Follow the procedure described in Chapter 1. (The file ACC.DIR must be modified to search the library MLIB4630 instead of MLIB.)

APPENDIX A

DIFFERENCES BETWEEN ACC AND STANDARD C

INTRODUCTION

The AMC implementation of C (ACC) differs in several ways from the C standard defined in The C Programming Language, by B.W. Kernighan and D.M. Ritchie. These differences fall into two categories, restrictions on the standard and extensions to the standard.

RESTRICTIONS

1. ACC requires that each external declaration be initialized exactly once in the set of files that comprise a C program. Standard C allows external declarations to remain uninitialized.

2. ACC requires that a macro's name and its arguments fit on one line. In standard C, a piece of text requiring macro expansion can span several consecutive lines.

3. ACC does not allow the use of the **sizeof** operator in #if constructions. This usage is allowed in standard C.

4. ACC requires that all parameters that might be passed to a function must be explicitly declared in the function body. (In other words, the maximum number of arguments must be specified.) Standard C allows functions with an unlimited number of arguments.

EXTENSIONS

1. ACC allows multiple-line text strings through the use of the backslash, newline convention. In standard C, this convention may be used only in strings.

2. ACC allows zero or more letters to be enclosed in single quotes (') and thereby converted to their ASCII binary codes. These binary codes are stored in bytes of memory. Standard C permits only single characters to be enclosed in quotes.

3. ACC will enforce strict member specification rules on the direct and indirect structure memory reference operators . and -> if so requested by use of the -m invocation option (see Appendix C).

4. A union may be initialized with ACC. The type of the first member of the union governs the interpretation of the initialization list. Standard C does not allow unions to be initialized.

5. The new types unsigned long and unsigned char have been added.

APPENDIX B

PROGRAMMING TIPS

Large source files sometimes present a problem for ACC. The size of a source file is a function of the number of symbols defined, the complexity of expressions, and the number of lines of code. In general, individual functions should be no more than a page or two in length (approximately 100 lines of code). Functions longer than this may cause the compiler's memory requirements to exceed the available storage on the AmSYS8/8, thus generating either a pass 2 error message or a system crash. (Good programming practice also suggests the use of small, easy-to-understand functions.)

APPENDIX C

INVOCATION PROCEDURE UNDER AMDOS

INTRODUCTION

The three passes of the compiler and the associated programs must be activated individually in the proper order, either manually by the user or by a command procedure submitted in the form of an submit file. The commands to activate these programs are the subject matter of the remainder of this appendix.

THE PREPROCESSOR

The preprocessor is used by the C compiler to perform #define, #include, and other functions signaled by a #, before actual compilation begins.

The command to invoke the preprocessor has the form

```
     accp0  option-string input-filename-string
```

accp0 is the preprocessor. It accepts several types of options and a sequence of one or more input source files identified by the given file names. The options and input filenames must be separated by one or more spaces.

The options are:

-c — Don't strip out /* comments */ nor continue lines that end with \.

-dname=def — Define name with the definition string def before reading the input; if =def is omitted, the definition is taken as "1". Embedded blanks must be enclosed in single quote marks. The name and def must be in the same argument unless the argument is quoted. Up to ten definitions can be entered in this fashion.

-i prefix — Add prefix to all files used in #include<filename>.

-o file — Write the output to the specified file and write error messages to console. Default is the console.

-pcharacter — Change the preprocessor control character from '#' to character.

-x — Put out tokens for input to the parser, not lines of text.

THE PARSER

Pass 1 is the parsing pass of the C compiler. It accepts a sequential file of tokens from the preprocessor and writes a sequential file of flow graphs and parse trees, suitable for input to the machine-dependent code generator. The operation of pass 1 is largely independent of any target machine.

The command to invoke the parser has the form

```
     accp1  option-string  input-filename
```

accp1 is the parser. It accepts input from the file identified by the given name.

The flag options are:

-m — treat each structure/union as a separate name space, and require x.m to have x a structure with m one of its members.

-o file — Write the output to the specified file and write error messages to the console. Default is the console.

-c — Map all external symbols into uppercase. Normally externals are case sensitive; local symbols always are.

THE CODE GENERATOR

Pass 2 is the code generating pass of the C compiler. It accepts a sequential file of flow graphs and parse trees from pass 1 and writes a sequential file of AmZ8000 assembly language statements, suitable for input to the MACRO8000 assembler.

As much as possible, the compiler generates free-standing code; but for those operations which cannot be done compactly, it generates inline calls to a set of machine-dependent library routines.

The command to invoke the code generator has the form

```
     accp2  option  input-filename
```

accp2 is the code generator. Input comes from the file identified by the given name.

The only option is

```
     -o  output-filename
```

This option directs the output from the code generator to the file named in the option and error messages to the console. If no output files is named, the console is assumed.

THE ASSEMBLER

MACRO8000 is the assembler. It accepts a file of AmZ8002 source code from pass 2 and produces a relocatable (.ZRL) output file.

The command to invoke the assembler has the form:

```
     MACZ  source options  overrides
```

MACZ is the assembler itself. Input comes from the identified source. The options are shown in table C-1. Overrides are explained in the MACRO8000 User's Manual.

TABLE C-1. MACRO8000 OPTIONS

NAME	DEFAULT	FORM	MEANING
Dots	No first pass listing	D	Produce a first pass listing that shows a dot for each statement assembled.
Error	Full program listing	E	Suppress full program listing and produce only a listing of errors.
Warning	No warnings	W	Enable listing of warning messages.
Listing	L=CON:	L	Send listing to dev:name.PRN on same drive as source, with same name as source.
		L=file	Send listing to the file dev:name.ext as specified.

		L=CON:
		L=LST:
			Send listing to console device (if printer is enabled with CONTROL P, listing also prints).

Object	No object file for input to LINK8000	O	Create object file dev:name.ZRL on same drive as source, with same name as source.
		O=file	Create object file dev:name.ext as specified.

THE LINKER

LINK8000 is the linker. It accepts .ZRL files from the assembler and the run-time library and produces an executable object (.BIN) file. The command to invoke the assembler has the form:

```
     LNKZ  dirfile options overrides
```

LNKZ is the linker itself. The input, output, and a number of other parameters are specified in the dirfile.

The options are shown in table C-2. The overrides are explained in the LINK8000 User's Manual.

TABLE C-2. LINKER OPTIONS

NAME	DEFAULT	FORM	MEANING
Listing	L=CON:	L	Send listing to dev:name.PRN on currently selected drive, with same name as dirfile
		L=file	Send listing to the file dev:name.ext as specified.
		L=CON:	Send listing to console device (if printer is enabled with CONTROL P, listing also prints)
		L=LST:	Send listing to printer device
Object (for MODULE, LIBRARY, ROMLIB run only)	No object file	O	Create object file dev:name.ZRL on same drive as dirfile, with same name as dirfile
		O=file	Create object file dev:name.ext as specified. The file type should not be $$$
Hex (for AmZ8002 PROGRAM only)	No hex file for PROM burning	H	Create hex file dev:name.HEX on same drive as dirfile, with same name as dirfile
		H=file	Create hex file dev:name.ext as specified
Binary (for PROGRAM run only	No binary file for downloading	B	Create binary file dev:name.BIN on same drive as dirfile, with same name as dirfile
ROMLIB (for PROGRAM run only)	No ROMLIB as linker output from RETAIN or OMIT	B=file	Create binary file dev:name.ext as specified
		R	Create ROMLIB dev:ROMLIB.ZRL on same drive as dirfile
		R=file	Create ROMLIB file dev:name.ext as specified. This file, which is called a ROM library index or ROMLIB, contains global symbol definition.

NOTE

When interactive input is specified (*) with options L, O, H, or B (without explicit filename), a default filename LINK is supplied in lieu of dirfile.

APPENDIX D

THE RUN-TIME LIBRARY

Table D-1 shows the functions that are described in this appendix. They are found in several different disk files and can be grouped into four categories, based on their implementation language and accessibility.

TABLE D-1. LIBRARY FUNCTIONS

	ACCSTD.H	CLIB	MLIB4630	MLIB	CINIT
MACROS	isalpha isupper islower isdigit isalnum isspace ispunct isprint iscntrl isascii	------	--------	--------	-------
C functions	-------	strcat strcat strcmp strncmp strcpy strncpy strlen index rindex atof atoi atol abs max min printf scanf puts gets	getchar putchar	--------	-------
C callable assembly language	-------	------	exit moncall (+MLIB)	inbyte outbyte inword outword	-------
assembly language	-------	------	C@ABORT	C@SAV C@SWITCH	C@INIT

This appendix covers the functions in library files, in the standard header file, and the CINIT file. Macros and most C language functions are described using a simple three-part structure:

1. The **NAME** subsection lists the names of the functions and gives a brief description of each function's purpose.
2. The **SYNOPSIS** subsection summarizes the definition of each function.
3. The **DESCRIPTION** subsection describes the function in detail.

Assembly language functions are documented by providing source code. (The source code of two C functions, **getchar** and **putchar** , is also provided.)

THE FILE ACCSTD.H

This file contains ten macro definitions. Macros are expanded by the preprocessor at compile time, so they cannot be called by assembly-language functions, nor can their addresses be taken with the & operator.

NAME:

isalpha, isupper, islower, isdigit, isalnum, isspace, ispunct, isprint, iscntrl, isascii - character classification

SYNOPSIS:

```
#include     <STD.H>
isalpha(c)
...
```

DESCRIPTION:

These macros classify ASCII-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. **isascii** is defined on all integer values; the rest are defined only where **isascii** is true and on the single non-ASCII value EOF.

isalpha	c is a letter
isupper	c is an upper case letter
islower	c is a lower case letter
isdigit	c is a digit
isalnum	c is an alphanumeric character
isspace	c is a space, tab, carriage return, newline, or formfeed
ispunct	c is a punctuation character (neither control nor alphanumeric)
isprint	c is a printing character, code 040(8) (space) through 0176 (tilde)
iscntrl	c is a delete character (0177) or ordinary control character (less than 040)
isascii	c is an ASCII character, code less than 0200

THE FILE CLIB.ZRL

CLIB.ZRL contains object code for 19 C-language functions. These functions can be divided into three groups: string handling functions, ASCII/Number conversion functions, integer functions, and I/O functions.

NAME:

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, index, rindex - string operations

SYNOPSIS:

```
char *strcat(s1, s2)
char *s1, *s2;

char *strncat(s1, s2, n)
char *s1, *s2;

strcmp(s1, s2)
char *s1, *s2;

strncmp(s1, s2, n)
char *s1, *s2;

char *strcpy(s1, s2)
char *s1, *s2;

char *strncpy(s1, s2, n)
char *s1, *s2;

strlen(s)
char *s;

char *index(s, c)
char *s, c;

char *rindex(s, c)
char *s;
```

DESCRIPTION:

These functions operate on null-terminated strings. They do not check for overflow of any receiving string.

strcat appends a copy of string s2 to the end of string s1. **strncat** copies at most n characters. Both return a pointer to the null-terminated result.

strcmp compares its arguments and returns an integer greater than, equal to, or less than 0, according as s1 is lexicographically greater than, equal to, or less than s2.
strncmp makes the same comparison but looks at, at most, n characters.

strcpy copies string s2 to s1, stopping after the null character has been moved. **strncpy** copies exactly n characters, truncating or null-padding s2; the target can not be null-terminated if the length of s2 is n or more. Both return s1.

strlen returns the number of non-null characters in s.

index (rindex) returns a pointer to the first (last) occurrence of character c in string s, or zero if c does not occur in the string.

NAME:

atof, atoi, atol - convert ASCII to numbers

SYNOPSIS:

```
double atof(nptr)
char *nptr;

atoi(nptr)
char *nptr;

long atol(nptr)
char *nptr;
```

DESCRIPTION:

These functions convert a string pointed to by nptr to floating, integer, and long integer representation respectively. The first unrecognized character ends the string.

atof recognizes an optional string of tabs and spaces, then an optional sign, then a string of digits optionally containing a decimal point, then an optional 'e' or 'E' followed by an optionally signed integer.

atoi and **atol** recognize an optional string of tabs and spaces, then an optional sign, then a string of digits.

NAME:

```
     abs - find absolute value of an integer
     max - find the maximum of two integers
     min - find the minimum of two integers
```

SYNOPSIS:

```
     int abs(i)
          int i:

     int max(il, i2)
          int il, i2:

     int min(il, i2)
          int il, i2:
```

DESCRIPTION:

abs returns the absolute value of its integer argument.

max determines and returns the greater of the two integer arguments.

min determines and returns the smaller of the two integer arguments.

I/O FUNCTIONS

NAME:

gets -- get a string from the console

SYNOPSIS:

```
char *gets(s)
char *s;
```

DESCRIPTION:

gets reads a string into s from the console. The string is terminated by a newline character, which is replaced in s by a null character. gets returns its argument.

NAME:

puts - put a string to the console

SYNOPSIS:

```
puts(s)
char *s;
```

DESCRIPTION:

Puts copies the null-terminated string s to the console and appends a new-line character.

This routine does not copy the terminal null character.

NAME:

scanf - formatted console input

SYNOPSIS:

```
scanf(format [, pointer ] ...)            - Limit of 10 arguments.
char *format;
```

DESCRIPTION:

scanf reads characters from the console, interprets them according to a format, and stores the results in its arguments. It takes as arguments a format specifier, described below, and a set of pointer arguments indicating where the formatted input should be stored. (No more than 10 arguments can be passed.)

The format specifier contains:

- Blanks, tabs, or newlines, which match optional white space in the input.

- A character (not %) which must match the next character of the input stream.

- Interpretation specifications, consisting of the character %, an optional assignment-suppressing character (*), an optional numerical maximum field width, and an interpretation character.

An interpretation specification directs the interpretation and assignment of the next input field; the result is placed in the variable pointed to by the associated pointer, unless assignment suppression was indicated by *. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The interpretation character directs the actual interpretation of the input field; the corresponding pointer argument must be of an allowed type. The following interpretation characters are legal:

% a '%' character occupies the input field;

d a decimal integer occupies the input field; the corresponding pointer must be an integer pointer.

o an octal integer occupies the input field; the corresponding pointer must be an integer pointer.

x a hexadecimal integer occupies the input field; the corresponding pointer must be an integer pointer.

s a character string occupies the input field; the corresponding pointer must be a character pointer pointing to an array of characters large enough to accept the string and a terminating '\0', which will be added. The input field is terminated by a space character or a newline.

c a character occupies the input field; the corresponding pointer must be a character pointer. The normal skip over space characters is suppressed in this case; to read the next non-space character, use '%1s'. If a field width is given, the corresponding argument should refer to a character array, and the indicated number of characters is read.

e
f a floating point number occupies the input field; the next field is converted accordingly and stored through the corresponding pointer, which must be a pointer to a float. The input format for floating point numbers is an optionally signed string of digits possible containing a decimal point, followed by an optional exponent field consisting of an E or e followed by an optionally signed integer.

The interpretation characters d, o, and x can be capitalized or preceded by l to indicate that a pointer to long rather than to int is in the argument list. Similarly, the interpretation characters e or f can be capitalized or preceded by l to indicate a pointer to double rather than to float. The interpretation characters d, o, and x can be preceded by h to indicate a pointer to short rather than to int.

The **scanf** function returns the number of successfully matched and assigned input items. This can be used to decide how many input items were found. The constant EOF is returned upon end of input; note that this is different from 0, which means that no interpretation was done; if interpretation was intended, it was not allowed because an inappropriate character appeared in the input.

Example:

The call

```
int i; float x; char name[50];
scanf( "%d%f%s", &i, &x, name);
```

with the input line

```
13  23.16-2  schmitlap
```

will assign to i the value 13, x the value .2316 and name will contain 'schmitlap\0'. Or;

```
int i; float x; char name[50];
scanf("%2d%f%*d%s", &i, &x, name);
```

with input

```
12345 6789 01b23 b
```

will assign 12 to i, 345.0 to x, skip '6789', and place the string '01623' in name. The next call to getchar will return 'b'.

NAME:

printf - formatted console output

SYNOPSIS:

```
printf(format [, arg ] ...)                - Limit of 10 arguments
char *format;
```

DESCRIPTION:

printf converts, formats, and prints on the console its arguments (those following the first argument) under control of the first argument. The first argument is a format specifier which contains two types of objects: characters, which are simply copied to the output stream, and interpretation specifications, each of which causes interpretation and printing of the next successive arg of **printf** . No more than 10 arguments can be passed.

Each interpretation specification is introduced by the character %. Following the %, there can be:

- an optional minus sign '-' which specifies left justification of the value in the indicated field;

- an optional digit string specifying a field width; if the value has fewer characters than the field width it will be padded with blanks on the left (or right, if the left-justification indicator has been given) to make up the field width; if the field width begins with a zero, zero-padding will be done instead of blank-padding;

- an optional period '.' which serves to separate the field width from the next digit specifies;

- an optional digit string specifying a precision which determines the number of digits to appear after the decimal point, for e- and f-formats, or the maximum number of characters to be printed from a string;

- the character l specifying that a following d, o, x, or u corresponds to a long integer arg. (A capitalized interpretation character has the same meaning.)

- a character which indicates the type of interpretation to be applied.

A field width or precision can be '*' instead of a digit string. In this case an integer argument supplies the field width or precision.

The interpretation characters and their meanings are:

d
o
x
The integer argument is interpreted as decimal, octal, or hexadecimal notation respectively.

f
The float or double argument is interpreted as decimal in the style '[-]ddd.ddd' where the number of d's after the decimal point is equal to the precision specification for the argument. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed.

e
The float or double argument is interpreted in the style '[-]d.ddde±dd' where there is one digit before the decimal point and the number after is equal to the precision specification for the argument; when the precision is missing, 6 digits are produced.

g
The float or double argument is printed in style d, in style f, or in style e, whichever gives full precision in minimum space.

c The character argument is printed. Null characters are ignored.

s The argument is taken to be a string (character pointer) and characters from the string are printed until a null character is reached or until the number of characters indicated by the precision specification is reached; however, if the precision is 0 or missing all characters up to a null are printed.

u The unsigned integer argument is interpreted as decimal and printed (the result will be in the range 0 to 65535).

% Print a '%'; no argument is used.

In no case does a non-existent or small field width cause truncation of a field; padding takes place only when the specified field width exceeds the actual width. Characters generated by **printf** are printed by **putchar.**

Example:

To print a date and time in the form 'Thursday, April 24, 11:45', where weekday and month are pointers to null-terminated strings:

```
printf("%s, %s %d, %02d:%02d", weekday, month, day, hour,
min);
```

To print to 5 decimals:

```
printf ("square = %.5f", 4*abs(3.14159));
```

THE FILE MLIB.ZRL

This file contains two classes of functions: port I/O functions and the C non-callable functions C@SAV and C@SWITCH.

PORT I/O

Low-level input and output (I/O) operations are provided by four assembly language functions. All of them require a user-specified port.

NAME:

inbyte, outbyte, inword, outword - port I/O functions

SYNOPSIS:

```
     int inbyte(port)
     unsigned port;

     int outbyte(port, c)
     unsigned port;
     char c;

     int inword(port)
     unsigned port;

     int outword(port, w)
     unsigned port;
     int w;
```

DESCRIPTION:

inbyte inputs a byte from the given port.

outbyte inputs a word from the given port.

outword outputs the given word to the given port.

The functions **inbyte** and **inword** return the byte or word fetched.

```
MODULE  'CPORTIO';
%       C RUN TIME ROUTINES
%       FOR 4016/4116 EVALUATION BOARD
%
        GLOBAL  INBYTE,
                INWORD,
                OUTBYTE,
                OUTWORD;
%
SEGMENT [@CUM], 'CODE';
%
%       EXECUTE IN(PORT)
%
INBYTE:
        LD      R1, R12^(2);
        INB     RL7, R1;
        CLRB    RH7;
        RET     ;
%
INWORD:
        LD      R1, R12^(2);
        IN      R7, R1;
        RET     ;
%
%       EXEC OUT(PORT, DATA)
%
OUTBYTE:
        LD      R1, R12^(2);
        LDB     RL7, R12^(5);
        OUTB    R1, RL7;
        CLRB    RH7;
        RET     ;
%
OUTWORD:
        LD      R1, R12^(2);
        LD      R7, R12^(4);
        OUT     R1, R7;
        RET;
%
        END.
```

C@SAV AND C@SWITCH

The assembly language functions C@SAV and C@SWITCH cannot be accessed from C programs; they are used by ACC itself. C@SAV is used to manage the frame stack and return stack; C@SWITCH is used to implement the switch - case control flow statement.

```
MODULE  'CSAVRET';
%       C RUN TIME ROUTINES
%       FOR Z8002
%
        GLOBAL  C@SAV,
                C@SWITCH;
        EXTERNAL C@ABORT;
%
SEGMENT [@CUM], 'CODE';
%
%       C FUNCTION ENTRY ROUTINE
%
C@SAV:
        LD      R1, R15^;
        LD      R15^, R12;
        ADD     R12, RO;                        %ALLOCATE NEW FRAME
        CP      R12, R15;                       %STACK OVERFLOW?
        JP      LLT, R1^;                       %JUMP IF NOT
        JP      C@ABORT;
%
%       SWITCH SELECT ROUTINE
%               R1 POINTS TO SWITCH TABLE, R7 HAS CASE VALUE
%
C@SWITCH:
        TEST    R1^;                    %DEFAULT CASE?
        JR      ZR, C@SWEND;            %JUMP IF SO
        INC     R1, 4;                  %BUMP TO NEXT CASE
        CP      R7, (-2)^(R1);          %THIS IT?
        JR      NE, C@SWITCH;           %JUMP IF NOT
        DEC     R1, 6;
%
C@SWEND:
        LD      R1, (2^)(R1);           %LOAD JUMP ADDRESS
        JP      R1^;
%
        END.
```

THE FILE MLIB4630.ZRL

This file contains four groups of functions. The first group is identical to the MLIB functions described in the previous section. The second group consists of the two functions **moncall** and **exit**, which are used to call or return to the 4630 Monitor. The third group consists of the functions **getchar** and **putchar** which are used for character I/O to the terminal. The final group consists of the single function C@ABORT, which may not be called from C programs, but is used to recover from stack overflows.

C@ABORT

This function can be dealt with very quickly. It is called by C@SAV whenever a stack overflows. When it is called, it executes a SC (System Call) instruction with parameter 99.

```
        MODULE  'CABORT';
%       STACK OVERFLOW ROUTINE
%
        GLOBAL  C@ABORT;
%
        SEGMENT [@CUM], 'CODE';
%
C@ABORT:
        SC      99;
        END.
```

NAME:
    moncall - do monitor service call

SYNOPSIS:

```
     struct monstr {
               char          mfunc,       /* function code */
                             mresp,       /* response */
                             *mfnp,       /* filename pointer */
                             *mstp;       /* string pointer */
               int           mstl;        /* string length */
               {;

     #define   MCRD          1            /* console read */
     #define   MCWR          2            /* console write */
     #define   MPWR          3            /* print */
     #define   MDOPN         4            /* open file */
     #define   MDCLS         5            /* close file */
     #define   MDCRE         6            /* create a file */
     #define   MDRD          7            /* read next sector */
     #define   MDWR          8            /* write next sector */

     moncall(rp)
     struct monstr *rp;
```

DESCRIPTION:
    The structure monstr describes the monitor's I/O control block (see Am96/4016 AmZ8000 Evaluation Board User's Manual). **moncall** loads its argument into register 1 and issues system call zero instruction. Function codes 3-8 are available only on systems with HOST.

    It returns the response code from the request block.

    The structure and function codes are defined in the standard include file "accstd.h".

NAME:

exit - terminate a process or program

SYNOPSIS:

```
exit (status)
int status;
```

DESCRIPTION:

exit terminates a program and, depending on the implementation and the system, may perform other termination duties.

```
        MODULE  'EXIT';
        GLOBAL  EXIT;
        SEGMENT [@CUM], 'CODE';
EXIT:
        SC      1;
        END.
```

NAME:

getchar - get a character from the console
putchar - put character to console

SYNOPSIS:

```
int getchar()

int putchar(c)
char c;
```

DESCRIPTION:

getchar obtains and returns the next character from the terminal.

putchar copies the character c to the console.

```
/* File:  getchar.c */

#include "accstd.h"
#define CTRLZ  0x1A

/*
          External routines in this compilation unit;

          getchar reads and returns character entered at console

*/

#define LINESIZE 80

static char inbuf[LINESIZE] {'\0'};
static int inptr = -1, incnt = 0;
static int eofset = 0; /* FALSE */

 char getchar() /* return a character buffered from console */

{

          if (eofset) return(EOF);
          if (inptr < 0 ||incnt <= inptr) { /* need to read new line */
                  incnt = getln(inbuf);
                  inptr = 0;
                  if (inbuf[0] == CTRLZ)
                          {
                          eofset = 1;    /* TRUE */
                          return(EOF);
                          }
          }
          return(inbuf[inptr++]);
}

static int getln(buf)
          char buf[];
{
          struct monstr rb;

          rb.mstp = buf;
          rb.mstl = LINESIZE;
          rb.mfunc = MCRD;
          moncall( &rb );
          buf[rb.mstl-1] = '\n';
          return( rb.mstl );

{
```

```
/* File:  putchar.c */

#include "accstd.h"

/*
          Routine in this compilation unit:

          putchar writes a character to the console

*/

char putchar(c)
register char c;

{

          char cbuf;
          struct monstr rb;

          if (c == '\n')
                  putchar('\r');

          cbuf = c;
          rb.mfunc = MCWR;
          rb.mstp = &cbuf;
          rb.mstl = 1;
          moncall( &rb );
          return(c);
}
```

THE FILE C@INIT

This file contains one function, C@INIT. This function sets up STACK@TOP and STACK@BTM and calls **main**.

```
MODULE  'C@INIT';
%       C STARTUP ROUTINE
%       FOR 4016/4116 EVALUATION BOARD
%
        GLOBAL  C@INIT;
        EXTERNAL STACK@TOP,
                STACK@BTM,
                MAIN;
        SEGMENT [@CUM], 'CODE';
%
%       C RUN-TIME ZERO ROUTINE
%
C@INIT:
        LD      R15, ^STACK@TOP;
        LD      R12, ^STACK@BTM;
        CALL    MAIN;
        SC      1;

        END.
```

APPENDIX E
ERROR MESSAGES

Error messages can be divided into four broad categories based on where they are generated: compiler messages, assembler messages, and run-time messages. In this appendix, only compiler error messages are documented. Assembler error messages can be found in the MACRO8000 User's Manual, linker error messages in the LINK8000 User's Manual. Run-time error reporting is the responsibility of the user program or the run-time monitor.

All three passes of the compiler generate error messages. These messages are described in the through sections that follow.

PREPROCESSOR ERROR MESSAGES

MESSAGE	MEANING (if not obvious)
bad #xxx	Unrecognizable # control line
bad #define arguments	Cannot parse #define line
bad flags	See Appendix B for p0 options
bad macro arguments	Cannot parse macro definition
bad output file	Cannot create output file
can't #include xxx	Cannot open file specified by #include
can't open xxx	Cannot open file specified in command line
illegal #if expression	
illegal #if syntax	
illegal ? : in #if	
illegal character: x	Not a recognizable token in C
illegal constant xxx	Not a recognizable numeric form
illegal floating constant	
illegal number in #if	
illegal operator in #if	
illegal unary op in #if	
line too long	More than 512 characters
misplaced #xxx	Preprocessor control line out of place
missing #endif	Unbalanced #if, #ifdef, or #ifndef
missing) in #if	
missing */	Unbalanced /* comment
string too long	More than 128 characters
too any -d arguments	More than 10 arguments
unbalanced x	When x is a delimiter, i.e. ', ", (, <, or {

PARSER ERROR MESSAGES

MESSAGE	MEANING (if not obvious)
arithmetic type required	Either integer or float necessary
array size unknown	
bad declaration	Contents of () unrecognizable
bad field width	Either negative or larger than word size (16 bits)
bad flag	See Appendix B for parser options
bad output file	Cannot create output file
cannot initialize	
constant required	
declaration too complex	More than five modifiers
function redefined: xxx	
function required	Arguments declared, but no function body
function size undefined	
identifier not allowed: xxx	
illegal &	Unary address of operator
illegal =+	
illegal assignment	
illegal bit field	
illegal break	
illegal case	
illegal cast	
illegal comparison	
illegal continue	
illegal default	
illegal double initializer	
illegal field	
illegal field initializer	
illegal indirection	Unary + operator
illegal integer initializer	
illegal member	
illegal operand type	
illegal pointer initializer	
illegal return type	
illegal storage class	
illegal structure reference	
illegal use of typedef: xxx	
integer type required	
lvalue required	
member conflict: xxx	Between structures
member redefined: xxx	Within a structure
missing argument	
missing expression	
missing goto label	
missing label: xxx	
missing member name	Identifier must follow . or ->
missing: xxx	Keyword or punctuation
no structure definition	
not an argument: xxx	
redeclared: xxx	

PARSER ERROR MESSAGES (continued)

MESSAGE	MEANING (if not obvious)
redeclared argument: xxx	
redeclared external: xxx	
redeclared local: xxx	
redeclared typedef: xxx	
redefined: xxx	
redefined label: xxx	
redefined tag: xxx	
structure size unknown	
undeclared: xxx	
unexpected EOF	
unknown member: xxx	
useless expression	Result unused, no side effect

CODE GENERATOR ERROR MESSAGES

MESSAGE	MEANING (if not obvious)
panic can't write	
panic FLOAT not implemented	
panic eval fail or lost regs	internal compiler error
panic excess refs	internal compiler error
panic too many refs	internal compiler error
panic can't do arg	internal compiler error
panic NO CC	internal compiler error
panic bad flag	
panic bad input file	
panic bad output file	
panic no FUNC	bad input file
panic BAD INPUT	bad input file
panic CHREAD	internal error
panic EOF	bad input file
panic funerr	internal error

APPENDIX F
RELEASE INFORMATION

The information contained in this appendix pertains to individual releases of the C Software Package.

RELEASE 1

1. External names are not case-sensitive, because in this release MACRO8000 and LINK8000 do no distinguish upper case letters from lower case letters. All external identifiers are shifted by the compiler to upper case.

2. MACRO8000 reserved words can not be used as external names. The words affected include SWAP, HIGH, LOW, SHR, SHL, AND, NOT, OR, EX XOR, MOD, CONST, and all the register names (Rn, RRN, RQn, RHn, and RLn, where n is an integer between 0 and 15).

3. Floating point arithmetic is not supported, so the float and double data types can not be used and **atof()** is not included in CLIB.

4. Multiplicative operators (*, /, %) involving unsigned long objects treat the operands as signed.

5. Underscores may not appear as the leading character of an identifier when used with V.2.0 of MACRO8000 or LINK8000.

INDEX

COMMENT SHEET

Address comments to:

Advanced Micro Computers
Publications Department
3340 Scott Boulevard
Santa Clara, CA 95051

TITLE: **AmZ8002 C SYSTEM INTERFACE**
PUBLICATION NO. 059910610–001 **REVISION A**

COMMENTS: (Describe errors, suggested additions or deletions, and include page numbers, etc.)

From: Name: ______________________________ Position: ____________________

Company: ____

Address: ____

Advanced
Micro
Computers

A subsidiary of
Advanced Micro Devices
3340 Scott Boulevard
Santa Clara,
California 95051
(408) 988-7777
TELEX: 171 142

ADVANCED MICRO COMPUTERS
3340 Scott Boulevard
Santa Clara, California 95051
Distributed by
Advanced Micro Devices