Rational
Systems, Inc.

P.O. Box 480
Natick, Mass. 01760

(617) 653-6194

May 10, 1985

Dear New Customer:

Enclosed is version 1.25 of *Instant-C*. The two diskettes (in the back pocket of the manual), contain:

- the *Instant-C* program,
- the library source files,
- a stand-alone version of the *Instant-C* editor,
- programs for configuring your keyboard and screen for the editor (if you don't have an IBM PC or compatible),
- a simple example program.

Also enclosed in this package are a complete manual and release notes for version 1.25. The release notes detail some new features that aren't in the manual yet. The manual will be updated and reprinted in the near future. You will automatically receive the updated version of the manual if you send in your registration.

You will want to read chapters 2 and 3 of the manual ("Overview" and "Getting Started with *Instant-C*") to most quickly learn how to use *Instant-C*. The interpreter environment is much different (and much better!) than the traditional tools for C you may have been using.

Version 1.25 is missing a few language features. All registered users will receive a **free** update to version 2.00 of *Instant-C*. Version 2.00 will handle larger programs and have auto initializers, bit fields in structures, and the ability to load .OBJ files created by other compilers or assemblers. The features will be added in approximately this order, and each feature may be available in a separate interim release. Please be sure to send the registration/user agreement to us, since we can't send you your **free** update(s) if we don't know where you are.

Instant-C is the <u>fastest</u> C interpreter, and it is the <u>best</u> environment for debugging C. If you find any problems in Version 1.25 that should be fixed or improvements that we can make, please let us know about them as soon as possible. Please take the time to tell us about your problems and suggestions. The feedback we have received from our early users has been a tremendous help to us in our work on *Instant-C*.

Sincerely,

Terence M. Colligan
President

encl.

Instant-C **Release 1.25**

This release of *Instant-C* contains many important improvements. The major areas are:

C Language Support	C pre-processor support is nearly complete, allowing defines with arguments. Also, static initializers are now supported. Details below.
Editor	Supports much larger files or buffers. Stand-alone editor reads files 2 to 4 times faster.
Library	Complete DOS 2.0 support (pathnames, devices, etc.). Other improvements include more run-time consistency checking, more functions, smaller/faster, improved compatibility with Lattice, C86, and other compilers. A math function library is included.
Debugger	Many improvements, including support for multiple screens while debugging (program output to one virtual screen with interpreter/debugger output to another).
Environment	Memory file management is more reliable. *Instant-C* makes better use of system memory.

C Language Support

Nearly all cases of static initializers (initializing declarations) are handled. The limitations are:

- initialized arrays must have the dimension specified, e.g., you must say

  ```
  long array[3] = {1, 2, 3};
  ```

  instead of

  ```
  long array[] = {1, 2, 3};
  ```

- comments are not handled in the initializer value list.
- only static variables may be initialized, not automatic variables.

General **#defines** are handled, but again with a few limitations:

- keywords and operators may not be **#defined**.
- **#define** text must be 'well-formed', i.e., may contain no unbalanced parentheses, brackets, or braces.
- you may not define pieces of declarations.

New Features

1. Static initialization of **long, float,** and **double** data is now supported.

2. Static initialization of pointer data is now supported.

3. Static initialization of **struct**'s is now supported.

4. Static initialization of arrays is now supported. Note, though, that *Instant-C* can't yet calculate the size of arrays from your initialization, so you must provide array dimensions for any array to be initialized.

5. The length and size of an argument list is now checked to match previous definitions or usage.

6. The **run** command now re-initializes all data to zero if there is no explicit initializer, or to the appropriate value if declared with an initializer.

7. **#define**'s with arguments are now supported.

8. Performance of operations on **float** variables is improved over version 1.01.

Editor

1. A "DOS" command is added, which, like **#shell** in *Instant-C*, gives you access to the operating system. You must be running on MS-DOS or PC-DOS 2.0 or later. This is available in the stand-alone editor as well as the *Instant-C* built-in editor. Remember to use the DOS **exit** command to return to the editor.

2. The editor command processor 'remembers' default arguments from one command to the next much better than in version 1.01. File reading from disk is much faster than in version 1.01.

3. A new function is available for the *Instant-C* editor. Function 51 is called "check, format, re-edit", and it does exactly that: your current buffer is compiled, and if compilation is correct, formats your function and leaves you in the editor. This makes it very easy for you to see the final form of your function, or to get it "tidied up" before you continue changing it. Note: if the

compilation is successful, the function or objects have been updated in the memory file. In the default PC keyboard configuration, ctrl-P is bound to this function.

4. The editor status line displays the bytes remaining available for buffers on the status line.

Instant-C Function Library

The library is distributed with DOS 2.0 file and device support. The library has a condensed source called LS1.C, and its components are defined in a file called LIB.IC.

An older "universal" library (which supports CP/M and DOS 1) is also on the distribution disks, but is not installed in DOS versions of IC.EXE. It is called LSU.C. The differences lie in STDIO.IC, which is replaced by the STDIODOS.IC file (for DOS 2). The DOS 2 library will not run on DOS 1 or CP/M-86 systems. LSU.C and STDIOU.H (rename it to STDIO.H) must be used for DOS 1 or CP/M-86.

The new library (LS1.C) has the following improvements over version 1.01:

1. Full support of pathnames;
2. Full support of devices;
3. Increased compatibility with libraries delivered with popular C compilers, plus conformance with proposed ANSI C standard.
4. More IO functions, e.g., **fread, fwrite, fseek, ftell, setbuf, rewind.**
5. The **fopen** function uses the standard convention for setting the file's mode. The general form is {rwa}[+][b], i.e., either "r", "w", or "a", (read, write, or append), optionally followed by "+" for update mode, optionally followed by "b" for binary (no new-line translations).
6. The new library is faster.

Both the DOS 2 and the previous "universal" library have other improvements as well:

1. More string functions (strncpy, strncmp, strncat);
2. The memory management functions check for more error conditions, including infinite loops in the free pool chains, **retmems** of previously returned areas, and overlaps of returned areas. This will make it easier for you to detect and debug memory management

problems.

3. **_main**, called to start your program for the **#run** command, performs redirection of **stdin** and **stdout** based on the < and > convention.

4. The functions **_inportw** and **_outportw**, which perform the 8086 IN word and OUT word instructions are now available.

5. A control-Z is now appended by **fclose** only to disk files opened for text output (i.e., not binary).

6. The file FUNCVAL.IC is deleted. The expression value display functions like **_int** and **_double** are now built into the debugger. However, you can still "roll your own" if you'd like: the internal display functions are used only if no user display function is declared.

   The debugger display functions are also internal with version 1.25. Just like the expression display functions, however, you can define your own **_pd**, etc., functions if you would like.

7. A new library source file, MISCLIB.IC replaces FUNCVAL.IC, and is combined in LS1.C and LSU.C. It contains **_main** and some other odds and ends.

8. A mathematics function library is delivered with version 1.25, called MATH.IC. It has the commonly available trig and transcendental functions, such as SQRT, EXP, SIN, COS, ATAN, etc. The header file ERROR.H defines the variable **errno**, which may be examined in the standard way for domain and range error conditions. A header file, MATH.H, declares the math functions to be of the appropriate type, usually double.

9. A header file FCNTL.H is provided to define certain values for the level-1 library, to be compatible with other compilers. For example, FCNTL.H contains a #define for O_RAW, which if or-ed or added to the mode parameter of **open** will suppress ascii newline translation (i.e., O_RAW is used for binary files). **open** in version 1.01 assumed binary mode, which was incompatible with some compiler's libraries.

Instant-C Environment

1. *Instant-C* version 1.25 requires *less* memory than version 1.01, but can use *more* if it is available. *Instant-C* will resize itself as you load and modify your programs, and uses memory only as needed. Versions 1.01 and before started up with fixed allocations of memory for various needs (source, user code, etc.), and these were sometimes imbalanced. Version 1.25 dynamically sizes these various segments as needed. One major improvement

mentioned before is the separation of editor buffers from *Instant-C* data, allowing much larger files, functions, or declarations to be edited. The "bad memory allocation" message bytes the dust.

2. A new built-in variable **_notabs** may be used to eliminate tab characters from *Instant-C* output. This can be set to 1 to remove tabs both from **#saved** source files and from the editor.

By the way, pending the next re-publication of the manual, here is a brief explanation of the built-in variables available, with default settings in brackets [ ]:

_intnum — [0xC0] defines the block of interrupts used by *Instant-C* -- may be redefined if there is a conflict with some other software/hardware (see pp. 146 and 161).

_stmcount — [10] defines the number of statements executed between checks for control-break interrupts -- may be set higher for slightly faster execution (at 10, it represents about 10% overhead). This does not affect the operation of the debugger or the **#step** command.

_remcol — [24] defines the first column that may be used by comments. Comments that start in column 1 remain in column 1, however. A larger column number may be used to move comments to the right, probably making a neater display at the cost of longer lines.

_tabwidth — [4] defines the spacing of tab positions. Eight is a very common alternative. If _notabs is 0, _tabwidth affects only displays by the #list and #type commands. Otherwise, it affects the expansion of tabs to spaces.

_tabindent — [0] this is the column number of the first tab position, or the left margin for source listings.

_notabs — [0] discussed above.

_screenlines — [24] defines the number of lines that may be output to the screen before a "more?" prompt and pause suspends output. May be set to 25 for 25 line screens, or to 0 for no pauses at all.

Treatment of special input characters is described below. Note: these are tentative assignments. We are likely to have configurable keyboard macros in a subsequent release.

ctrl-C, ctrl-X, or ESCape — clear entire line of input.

Ctrl-H, Backspace, Del, or Rubout
erase last character entered.

F1 or ctrl-N copy 1 character from corresponding position in last line entered.

F3 or ctrl-R copy remaining characters from last line entered.

And some frequently used debugger commands are built-in:

F10 or ctrl-J or newline
the **#step** command.

F8 or ctrl-I the **#step in** command.

F6 or ctrl-O the **#step out** command.

Instant-C Debugger

More run-time checking is provided in version 1.25. In particular, function returns validate the return address, before possibly returning to oblivion via a smashed return pointer. This can happen if store through bad pointers or off the end of an array. Array bounds checking, pointer accesses, and indirect function calls will be checked in the next version.

1. Several fixes and improvements are made in the **#step** commands in version 1.25. **#step** *count*, where *count* is an integer constant, will cause your program to continue for *count* more statements, and then breakpoint. Of course any exception, explicit breakpoint (_() function), or tracepoint will stop execution before hand.

2. At the completion of a #step operation, the completed statement source is displayed (as in 1.01). With 1.25, much more sophisticated look-ahead is performed to find the next statement that will be executed upon resumption. This is indicated by "next>". If the next statement does not sequentially follow the completed statement ("if( )", for example), a message such as "to line 99" is displayed, and then the next line is shown. Version 1.01 did not do this look-ahead, although it would show a "next>" line.

3. The command **#pu**, for print unsigned, is now implemented. It was documented in version 1.01 but not really available.

4. As discussed in Library, above, immediate mode expression evaluation display is handled internally. In version 1.01, library functions (which you could modify), with names of the form **_int** for integer values or **_double** for double precision, handled the display. These are no longer necessary, but if they are

present they will be used, so you can still customize your debugging and evaluation displays. The debugging display functions (with names like _pd for decimal output) are also internalized, but will be used instead of the internal code if you have defined them.

5. Debugger display functions **#pf** and **#plf** are now available for the display of **floats** and **doubles**, and work like the other such commands (see **#px** for example).

6. Source code displayed in **#back**traces and after steps is truncated at 80 columns for more consistent and readable displays. Of course, you can still see full text with the editor or the **#list** command.

7. Temporarily, there is a bug such that blank lines between functions may not always be properly retained between **#loading** and **#saveing**. The results are not horrible, and you may not even notice the problem. In any event, the results represent an improvement over 1.01.

Summary

We want to hear about any problems you may encounter, any improvements you may suggest, and, certainly, any successes that you have with *Instant-C*. As always, if you like *Instant-C*, tell your colleagues; if you don't like it, tell us.

Instant-C™ User's Manual

Instant-C User's Manual

(Version 1.01)

Terry Colligan

Ben Williams

Rational Systems, Inc.

Natick, Massachusetts

December 31, 1984

Instant-C User's Manual

Disclaimer

Rational Systems, Inc. makes no representations or warranties, either expressed or implied, with respect to the adequacy of this documentation or the programs which it describes in regard to merchantibility or fitness for any particular purpose or with respect to its adequacy to produce any particular result. The computer programs and documentation are sold "as is," and the entire risk as to quality and performance is with the buyer. In no event shall Rational Systems, Inc. be liable for special, direct, indirect or consequential damages resulting from any defect the programs, documentation, or software. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, in which case the above limitations and exclusions may not apply to you.

CI-86 is a trademark of Computer Innovations, Inc.

CP/M is a registered trademark of Digital Research, Inc.

Instant-C is a trademark of Rational Systems, Inc.

Intel is a trademark of Intel Corporation

Lattice-C is a trademark of Lattice, Inc.

MS-DOS is a trademark of Microsoft, Inc.

PC-DOS is a trademark of IBM.

UNIX is a trademark of Bell Telephone Laboratories, Inc.

Table of Contents

User's Manual

Instant-C™

Chapter 1

Introduction

Welcome to *Instant-C* (tm), an optimizing interpreter which will make your C language programming faster, easier and simpler than ever before. It operates on Intel 8086, 8088, and compatible microprocessors, under the PC-DOS, MS-DOS, CP/M-86, or MP/M-86 operating systems. Version 1 supports all standard C language features except for: initialization, parameterized **#defines**, declarations in compound statements, bit fields, a general assembly language interface, and certain obsolete operators. It includes the Unix Version 7 C compiler **void** data type for non-valued functions. The function library provided with *Instant-C* is designed to be compatible with UNIX Version 7 from Bell Laboratories, and with other C compilers for the 8086, particularly the Lattice-C and CI-86 compilers.

In addition to the normal features of any C compiler, *Instant-C* provides a unique programming environment which will greatly improve your productivity while creating or enhancing C language programs. This improvement occurs because we have made the edit-compile-run cycle the shortest possible, often less than two seconds.

Instant-C is not only effective for developing and enhancing C programs, but is also the best way to the C language.

You can run *Instant-C* on an IBM PC or compatible computer with at least 320K of memory. *Instant-C* will run under any of PC-DOS, MS-DOS, CP/M-86, MP/M-86 or CCP/M-86. You should have at least 280K available for programs on your system. *Instant-C* can use up to 440K of memory if it is available. You need one floppy disk with at least 240KB capacity to get started.

This manual assumes that you already know C. We do not attempt to teach you the language. If you are a beginner to C but know Basic, we suggest that you get a copy of the C Programming Guide by Jack Purdum (Que Corporation, 1983). Many of the programs in the C Programming Guide have been tested in *Instant-C*.

If you are a beginner to C, but don't know Basic (or think Basic is a mistake that should have been corrected long ago), we suggest that you get a copy of The C Programming Tutor by Leon A. Wortman and Thomas O. Sidebottom (Robert J. Brady Co., 1984).

This *Instant-C* User's Manual does not completely define the C language, nor does it serve as a reference for C. You should have a copy of The C Programming Language by Brian W. Kernighan and Dennis M. Ritchie (Prentice-Hall, 1978). We will refer to this book several times in the text as "K&R".

Instant-C is an ambitious undertaking; it is the first system of its kind. In any software of this complexity, especially in one so new, there are likely to be bugs. To reach our objective of eliminating all bugs from *Instant-C*, we need your help. Please report any problems, inconsistencies, or inconveniences you encounter. Appendix G provides details on how to do so. We will greatly appreciate any help you can provide.

You have purchased the most effective tool for developing and enhancing C language programs. We hope that you will enjoy using it.

Conventions Used in this Manual

All examples are set off and indented. If an example is interactive (as opposed to the calling sequence of a library routine), we show what the user types by using *italics like this*.

Chapter 2

Overview of *Instant-C*

This chapter provides a framework for your understanding of *Instant-C*, so that you can make better use of the information in the following chapters. It contains an overview of what the various parts of *Instant-C* are and how they work together.

Instant-C is a totally new kind of programming environment for the C language. It was inspired by a number of computer science research systems such as MACLISP, INTERLISP, and SMALLTALK. (If you are familiar with these systems, you can probably skip the rest of this chapter.)

We designed *Instant-C* to greatly speed up the edit-compile-load-test cycle in which many C programmers spend most of their time. *Instant-C* can be as much as 100 times faster than the traditional C language tools.

2.1 Components of *Instant-C*

Instant-C contains a number of components that correspond to tools in traditional programming environments:

C Compiler Converts C source language text into executable machine instructions.

C Interpreter
Runs C programs interactively. (Not generally available in other products.)

Full Screen Editor
Creates and modifies the C source language text.

Linker/Loader
Combines functions from multiple source files into a single program.

Pretty Printer
Reformats C source language text so that it has a standard, easy to read and understand, layout. (Not generally available in other products.)

C Function Library
Provides pre-written versions of commonly used operations, such as disk file reading or writing.

Source Language Debugger
Helps debug and trace C programs and examine and modify both programs and data. (Not generally available in other products.)

System Checker (LINT)
Checks that C programs made from multiple source files are consistent. (Not generally available in other products.)

In *Instant-C*, all of these components are combined into a single, unified system which handles all of your programming needs for the C language.

2.2 Organization of *Instant-C*

You can think of *Instant-C* as two different cooperating programs, the interpreter and the editor, which automatically and invisibly invoke all of the other components as needed to perfect your program, which you might view as a third co-operating program. You switch back and forth between the editor and the interpreter and don't even have to think about the other tools that are helping to speed your programming.

Since you will usually be using *Instant-C* to work on programs which are interactive (read from your keyboard and write to your screen), you actually have three programs to interact with: the *Instant-C* interpreter, the *Instant-C* editor, and your own program. You can think of these programs as three different parts of the total *Instant-C* environment:

1. **Interpreting**
   The interpreter reads your commands and executes them, either directly or via the debugger. Some of the interpreter commands switch you to the editor and others will switch to your program. Generally, the interpreter acts on your input one line at a time.

2. **Editing** The full-screen editor reads your command characters and manipulates C language source text. Some of the editor command characters switch you back to the interpreter. Generally, the editor acts on your input one character at a time.

3. **Executing Your Program**
   Your program can switch back to the interpreter by calling the **exit** function. *Instant-C* will also automatically switch back to the interpreter if your program makes an error, such as dividing by zero, or if it encounters a breakpoint. The style of interactions when your program is in control obviously depends upon your program.

Although you will interact with only one of these programs at a time, you can switch rapidly and easily between them. Since there is a different style of interaction in each program, you should understand how they are different to minimize confusion. The differences between these three programs are described in more detail below.

2.3 How the Pieces Fit Together

The following diagram shows how the three basic pieces of *Instant-C* fit together, and how you can switch between them. (The labels on the arrows will be explained in the following text.)

```
    Operating System
       |          ^
    ic|name       |  #quit
       |          |
       |          |
       V          |        #ed name
    ______________|_                       _______________
   |                |-------------------->|               |
   |   Instant-C    |                     |  Instant-C    |
   |  Interpreter   |  Ctrl-F (No errors) |    Editor     |
   |  & Debugger    |<--------------------|               |
   |________________|  Ctrl-Q (Quit)      |_______________|
          |       ^
  #run,   |       |
  funct() |       | return, exit, _exit (in your program)
          |       | (breakpoint encountered)
          |       | (runtime error)
          V       |
    +-----------------+
    |      Your       |
    |    Programs     |
    + - - - - - - - - +
    |    Instant-C    |
    |    Libraries    |
    +-----------------+
```

Notice that the interpreter is the central controlling piece, invoking either your programs or the editor to carry out your commands.

This information is explained in more detail below. For each of the three environments, we explain when to use it, how you can tell that you're in it, how you can get into it, and how you can get out.

2.3.1 The Interpreter

You can tell that you're interacting with the interpreter by the "# " prompt it uses and by the full line style of input. (The interpreter starts executing your commands when you press the **RETURN** key.)

You use the interpreter for the overall control of *Instant-C*, to load and save your programs, to test your programs, to set breakpoints, to display data, and to invoke the editor.

Enter the interpreter by typing the **IC** command to the operating system, by returning from the editor, or by returning from your program.

You can get out of the interpreter by typing the **quit** command. (But make sure you save your work first! -- see the **#save** command in the Command Reference Chapter.)

2.3.2 The Editor

You can tell that you're interacting with the editor by the distinctive screen format with two status lines at the top of the screen, by the lack of the "# " prompt, and by the single character style of interaction.

You use the editor to create C source text or to fix or enhance existing C source text. (That is, to modify or create your programs.) You will most frequently edit a single function. You can also edit global data or **#define** declarations. The editor always works on a copy of your functions or declarations. The place where the copy is stored is called the current buffer.

Enter the editor by typing the **#ed** command to the interpreter, or when the compiler finds a syntax error in a C language source file that you requested *Instant-C* to process.

You can leave the editor and go back to the *Instant-C* interpreter by typing the command **F** (**Ctrl-F** on most keyboards), which will cause *Instant-C* to try to compile the contents of the editor's current buffer. If the compiler finds no errors, your functions will be updated, and you will be returned to the interpreter. If the compiler finds an error, the error message will be displayed on the top line of your screen, and you will be left in the editor.

You can also leave the editor by typing the command Q (**Ctrl-Q** on most keyboards). In this case, the contents of the editor buffer are discarded and you are returned immediately to the *Instant-C* interpreter. The C functions and/or declarations you were editing are not updated in this case, since just the copy in the editor's buffer is discarded.

2.3.3 Executing Your Programs

You execute your program either to try it out, to test it, or to use it.

You can normally tell that you are executing your program because it will act differently than either the interpreter or the editor. (Your program may have a distinctive style of interaction and be easily recognizable.)

You start executing your program by typing a valid function call while you are in the interpreter. For example:

    # *main();*

or :

    # *printstatus(3)*

You can stop executing your program and get back to the interpreter in the following ways:

1. Your program calls the function **exit** or **_exit** directly.
2. The function that you called from the interpreter executes a **return** statement, or

reaches the end of the function.

3. A breakpoint you set is encountered during the execution of your program.

4. A function that you have traced is called or returns.

5. Your program completes a statement while you are single stepping through your program.

6. Your program makes a runtime error, such as division by zero.

7. You interrupt your program by typing **Ctrl-Break** or **Ctrl-C**.

2.4 Style Differences

Since *Instant-C* is such an advance over the previous generation of software tools for the C language, some people have been confused by the differences in style. Here are the most common areas of confusion with some (hopefully) clarifying explanation.

1. The most common thing to edit in *Instant-C* is a function; in traditional environments, the only thing that you can edit is a disk file. Since disk files must be loaded and then edited a function at a time, a very common error is to try **ed filename**, which doesn't work in *Instant-C*. For more information, see the section on Source File Handling, below.

2. If you have worked with other C tools, it may make you nervous not to explicitly compile your program. *Instant-C* automatically compiles each function when you leave the editor. All programs are always kept in compiled form. This is further explained in the next section.

3. Similarly, there is no explicit loading or linking for you to worry about. *Instant-C* is a

compile-to-memory system, and bypasses the need for a separate linker or loader. The compiler does all of the linking necessary during compilation.

2.5 What if I Type Something Incorrectly?

Correction of keyboard input errors will differ depending upon whether the interpreter, the editor, or your program is in control. On MS-DOS and PC-DOS systems, the interpreter uses the operating system's input editing. On the IBM PC and compatibles, the ESC key cancels the current line, and the function keys allow you to edit and re-enter a previous line.

On CP/M-86 and compatible systems, an interpreter command line can be cancelled at any time before you hit **RETURN**. It is cancelled by entering a **control-C**, **control-X**, or **ESC**; the characters previously input will be cleared away and the "# " prompt redisplayed.

On CP/M-86 based systems, the **backspace**, **control-H**, and **del** keys will erase the last character on the input line. We chose these character to match the system or programs you may be used to.

The editor has many ways to correct input, and they are detailed in the chapters on the editor.

Methods for correcting input to your program will, of course, depend on the program and the library functions involved.

2.6 Interpreter Output

In general, information displayed by the interpreter is output one line at a time. If twenty-three lines have been printed on the screen since the last keyboard input, however, the output will cease and a special prompt will appear **"more? (control-C to abort)"**. Output will resume and the "more?" prompt will be erased, when you hit any key except **control-C**. If you don't want to continue, you can return from command execution directly to the *Instant-C* interpreter by entering a **control-C**. (This pagination of interpreter output is controlled by the **_screenlines** system variable.)

Output from the *Instant-C* interpreter can be directed to your printer if you wish. See the **outfile** command in Chapter 6.

2.7 Where Are All Those Tools?

Earlier, we listed all of the components of *Instant-C*. We then described in detail the only two that you interact with -- the interpreter and the editor. These two programs create an environment in which all of the other tools are automatically invoked as needed. The following is a repeat list of those tools, with an indication of which tools are automatically invoked, and when. We hope this helps to dispell some of the mystery of *Instant-C*.

C Compiler
: (Invisible Tool) Automatically invoked when you give the save command (**Ctrl-F**) from the editor or when you execute a **#include** or **#load** command in the interpreter. The compiler is only noticeable when you save a large function because of the slight delay for it to finish its work.

C Interpreter
: (Main Environment) The "Control Center" of *Instant-C*. Moves you between environments, as well as doing the more traditional job of executing the C statements and expressions you type.

Full Screen Editor
: (Second Environment) The main way in *Instant-C* to change or create your

programs.

Linker/Loader
(Invisible Tool) Automatically invoked as part of the (invisible) compilation process.

Pretty Printer
(Invisible Tool) Automatically invoked as part of the editing process.

C Function Library
These routines are included as part of *Instant-C*. They are not exactly invisible, but rather passively wait for you or your program to call them.

Source Language Debugger
This tool is integrated with the interpreter. It is visible only as several extra commands in the interpreter.

System Checker (LINT)
(Invisible Tool) Automatically invoked when you do a save command in the editor or when you load a disk file by typing an **#include** or **#load** command. You see the system checking as additional error messages exactly like the normal syntax error messages.

2.8 Source File Handling

Most C programs consist of multiple source disk files. Because we designed *Instant-C* to be compatible with existing C programs and compilers, we have added a number of commands to deal with multiple simultaneous source disk files. Since no other interpreters for any language deal with multiple source files, the following overview of disk file handling should help you understand what is happening.

2.8.1 Structure of C Source Disk Files

In order to understand how *Instant-C* deals with disk files, you should understand how the systems with which we are trying to be compatible organize disk files. (If you are an experienced C programmer, you may want to skip the rest of this section.) In traditional C compiler environments, your program will consist of multiple source disk files. There are normally two kinds of disk files:

Source Disk Files
These contain the function definitions for your program. (The function definitions are all of the executable code in your program.) There may also be data declarations in them, but normally, source disk files consist of mostly function definitions. The names for these disk files usually end in **".C"**.

Header Disk Files
These contain data declarations and **#define** definitions that are shared by multiple source disk files. Each header disk file is invoked by one or more source disk file with an **#include** statement. The names for header disk files usually end in **".H"**.

2.8.2 The *Instant-C* Workspace

To be completely compatible with existing C language systems and existing C programs, *Instant-C* simulates multiple disk files by organizing its symbol tables into separate tables, one for each source disk file and one for each header disk file. Since each disk file inside *Instant-C* will have a symbol table with the same name, you can easily become very confused trying to understand whether something happens on disk or in the *Instant-C* work space. To help minimize that confusion, we will from now on always refer to files on disk as "disk files", and the corresponding symbol table inside *Instant-C* as "memory files".

To run a C program with *Instant-C*, you need to load disk files into memory in the *Instant-C* workspace with the **#load** command. You give one **#load** command for each source disk file in your program. The header disk files are loaded automatically, as specified by the **#include** statements in your source files. (You can also explicitly add a header disk file with a **#load** command, of course.)

Instant-C always has one memory file designated as the **current memory file.** Any new declarations or functions you type in the interpreter are added to the end of the current memory file. Since you can look at local data in the current memory file, you will probably need to switch back and forth between memory files. The **#use** command does exactly that. The **#use** command can also just display what the name of the current memory file is.

If you wish to create a new memory file, the **#new** command will do so. In addition, you can give the **#new** command to clear out the current memory file if you wish to start over.

Finally, to store your changes permanently on disk, you need to move your source code from its memory file(s) to the corresponding disk file(s). You use the **#save** command to write memory files to disk files. The **#save** command writes the current memory file to a named disk file.

2.8.3 File Summary

In summary, each disk file of C source language has a memory file corresponding to it in the *Instant-C* workspace. The following commands operate on memory files:

load, #load Reads the disk file contents into the memory file.

save, #save Writes the current memory file to a disk file.

use, #use Makes a memory file be the current memory file.

new, #new Creates a new memory file, or clears an existing memory file.

Chapter 3

Getting Started with *Instant-C*

This chapter tells you how to install *Instant-C*, how to use the small example program included with the package to test *Instant-C*, and how to start using *Instant-C* by typing commands to the interpreter.

3.1 System Requirements

Instant-C is designed to run on an IBM PC or PC/XT computer with at least 320K of memory. *Instant-C* will run under any of PC-DOS, MS-DOS, CP/M-86, MP/M-86 or CCP/M-86. (Multi-user or multi-tasking operation systems such as MP/M-86 or CCP/M-86 may require significantly more memory.) You should have at least 260K available for programs on your system. Because of the space your operating system takes, you will need at least 320K in your machine. *Instant-C* can use up to 440K of memory if it is available.

You do not need much disk capacity to run *Instant-C*; it will run on a single 320KB disk drive. Since *Instant-C* does not use the disk except to read and write your disk files, you can start up *Instant-C*, and then replace the *Instant-C* disk with a different disk containing your own C programs.

Instant-C requires Version 1.25 or later of PC-DOS or MS-DOS.

Instant-C works well on hard disks; it has no copy protection requiring that it reference any floppy disk.

3.2 Backing Up the *Instant-C* Disk

Before starting to use *Instant-C*, you should make at least two working copies of the distribution diskettes. One copy is for backup; the distribution diskettes can serve as a second backup. Since *Instant-C* is sometimes distributed on single-sided and/or single-density diskettes, you may want to change the format of the copies to make more space available for your programs. If so, you should not use DISKCOPY to make your backup. Rather, you should use COPY to copy the disk files from the *Instant-C* disk to your backup disk.

If you are using *Instant-C* on a computer with only a single floppy disk drive, you will probably want to install your operating system on the disk so as to make your *Instant-C* disk "bootable".

If all this is incomprehensible to you, or if you have never made a backup disk before, please go get some help from your dealer or a knowledgeable friend. We haven't provided enough introductory material here to teach you.

3.3 Running the Test Program

To confirm that you have successfully installed *Instant-C* onto your working disk, and verify that *Instant-C* can handle existing C language source disk files, we have included a test disk file, HELLO.C, on your distribution disk. This program prints on your screen the words:

```
Hello, World!
```

(This is the very first program described in K&R.) If you display the disk file HELLO.C with the operating system command **type**, you will see the following C program:

```
main()
{
    printf("Hello, World!\n");
}
```

Running the test program requires only three steps:

1. Start up *Instant-C*.

2. Add the program to the *Instant-C* workspace. (The workspace contains a copy of the programs or functions that *Instant-C* is currently handling.)

3. Tell *Instant-C* to execute this program.

Starting up *Instant-C* is easy. Make sure that you have your *Instant-C* working disk inserted in the computer. Then, type the command:

A>*ic*

After the operating system has found and loaded *Instant-C*, the following messages will be displayed:

Version 1.01, December 31, 1984
Copyright (C) 1984 by Rational Systems, Inc.

#

The "# " prompt indicates that you are in *Instant-C*'s interpreter and it is ready to process your commands. (The "# " was chosen because *Instant-C* command level is somewhat like the preprocessor input of a traditional C compiler.)

To load HELLO.C into your *Instant-C* workspace, type the command:

\# *load "hello.c"*

Instant-C will respond with the message:

main defined.
#

telling you that it has completed compiling main. If there had been any syntax errors in the disk file, *Instant-C* would have automatically switched you to the built-in editor so that you could correct the error.

Then, to run this program, simply enter the C language phrase:

```
# main()
```

Instant-C will respond with:

```
Hello, World!
#
```

When you are finished with your *Instant-C* session, use the quit command to return to the operating system:

```
# quit

A>
```

You can also give a filename when you start up *Instant-C*. This has the effect of automatically executing the **#load** command. If you type the command:

```
A> ic hello
```

Instant-C will respond with:

```
Version 1.01, December 31, 1984
Copyright (C) 1984 by Rational Systems, Inc.

main defined.
#
```

At this point you can execute main or quit, just as in the previous case.

3.4 Trying the Interpreter

To gain some familiarity with how *Instant-C* works, try interacting with the interpreter.

Start up *Instant-C* as before: Make sure that you have your *Instant-C* working disk inserted in the computer; then type the command:

```
A>ic
```

After the operating system has found and loaded *Instant-C*, the following messages will be displayed:

```
Version 1.01, December 31, 1984
Copyright (C) 1984 by Rational Systems, Inc.

#
```

Now, to try some interactions, type:

```
# 17
```

Instant-C should respond with:

```
          17
#
```

Each time you type a valid C language expression, *Instant-C* will evaluate it and display its value in an appropriate format. (17 is a very simple expression.)

Now, try something a bit more complicated:

```
# 3+4
```

You will see:

```
          7
#
```

Or:

```
# 5 * (3+4)
```

You will see:

```
          35
#
```

Integers are displayed as decimal numbers, even if they started as something else:

```
# 0xff
```

(0xff is 'ff' in hexadecimal.) *Instant-C* will print:

```
          255
#
```

Similarly, octal constants (leading 0 with no 'x') are converted to decimal:

```
# 0123
          83
#
```

You can try some of the fancier C operators ("<<" is left shift):

```
# 3 << 2
          12
#
```

You can type things other than integers:

```
#  "This is much easier than a compiler!"
```

(The double quotes define a string constant.) You should see:

```
          "This is much easier than a compiler!"
#
```

You can also type char constants:

```
# 'a'
          'a'
```

```
#
```

Finally, you can call functions (**toupper** is a library function that converts a char to upper case):

```
# toupper('q');
      'Q'
#
```

Try other expressions to explore how C operators work and to gain confidence in interacting with the interpreter. When you are finished, you can return to the operating system by typing:

```
# quit

A>
```

You have compiled and run a program (HELLO.C) and have invoked a function (**toupper**) directly. See how easy it is to use *Instant-C*? Read the next chapter to learn how to create new programs with *Instant-C*.

Chapter 4

Using the *Instant-C* Editor

This chapter is an introduction to the editor in *Instant-C*. It includes enough information so that you can successfully edit any declaration or function, but it does not cover all of the features of the editor.

4.1 Creating new functions

In *Instant-C* you create new function with the editor. To create a new function, simply type the command:

\# *ed funname*

where *funname* is the name of the function you wish to create. *Instant-C* will shift to the editor, and will prepare a template of a function definition for you. *Instant-C* can't distinguish between disk file names and function names. DO NOT use the name of a disk file instead of the name of a function -- you can't edit a disk file with the editor in *Instant-C*.

To insert text, use the arrow keys to move the cursor to the place on the screen where the text should go. The editor will already be in insert mode. In this mode, any normal characters (not control or function keys) that you type go onto the screen and into your C program.

If you wish to delete characters, move the cursor to the first character to delete and press the **Del** key. The character will disappear, and the screen will be updated appropriately.

When you are finished defining the function, type **Ctrl-F** to have your function automatically compiled. If there are no errors in your function, your function will be formatted and you will be returned to the *Instant-C* interpreter.

If, however, the compiler finds an error in your function, you will be left in the editor, in insert mode. The cursor will be placed at the point in your program where the compiler discovered the error, and an error message will be displayed on the top of the screen.

After correcting the error by inserting and deleting characters, you can again try to compile by typing **Ctrl-F**. You can repeat this cycle as many times as you wish until you have successfully compiled your program.

If you decide to abandon the editing you are doing, type **Ctrl-Q** instead of **Ctrl-F**. You will return to the *Instant-C* interpreter, but what you were editing will not be compiled, nor will any of your programs be updated. Your programs will be in the same state they were in before you started editing.

If you don't have time to finish getting all of the errors out of your program, you have three choices: The first is to simply put comments around the erroneous text, and then save the program normally. The second is to use the **Write** command in the editor's command mode -- see Chapter 8 for more details. The third is to use **#if 0** and **#endif** to conditionally omit the portion in error.

4.2 Modifying Existing Functions

To make a change or improvement to an existing C function that has already been loaded into the *Instant-C* workspace, you use the same process. Type the command:

 # ed funname

where **funname** is the name of the function you wish to modify. (Again, this is the name of a function and not a disk file name such as HELLO.C). The current definition of the function will appear on an editor screen. The editor will be in insert mode, so that you can use the same insert and delete process and the same **Ctrl-F** and **Ctrl-Q** mechanisms to control your modifications.

Since you often will be editing the same function repeatedly, *Instant-C* allows you to edit again the last function you were working on by omitting the name of the function to edit. That is:

 # ed

will edit the last function you were editing. If you omit the function name argument the first time that you use the editor, *Instant-C* will print an error message.

To edit disk files outside of *Instant-C*, see Chapter 7 on using the stand-alone version of *Instant-C*'s editor.

See Chapter 8 for more details on the *Instant-C* editor.

Chapter 5

Running Programs

This chapter covers all of the commands in *Instant-C* and tells you how to run your programs.

5.1 Executing Expressions

Whenever you are interacting with the *Instant-C* interpreter, you can enter any valid C expression for immediate evaluation. *Instant-C* will execute your expression and display the resulting value. For example,

```
# 2+3*4
```

will result in the following display:

```
14
```

Any external or global data variables can be used, as well as any static data variables in the current memory file if they are declared outside of any function. For the purpose of recognizing names, your expression is treated as though it were in a function at the bottom of the current memory file.

Instant-C converts your expressions into calls on some built-in functions, and those functions actually do the displaying. Different functions are called depending upon the data type of the expression. The specific functions called to to display the values of expressions are:

_char	Display the value of **char** expressions. (Note that there are relatively few **char** expressions, since **chars** are expanded to **ints** if they are combined with anything else in an expression.)
_short	Display the value of **short** or **short int** expressions. (Similarly, there are relatively few **short** expressions.)
_int	Display the value of **int** expressions.
_unsigned	Display the value of an **unsigned** or **unsigned int** expression.
_long	Display the value of a **long** or **long int** expression.
_float	Display the value of a **float** expression. (Note that there are few **float** expressions, since **floats** are expanded to **doubles** in most contexts.)
_double	Display the value of a **double** expression.
_string	Display the value of a **char** pointer expression (something declared **char ***).
_ptr	Display the value of all other pointers as a hex value.

In the preceeding example, *Instant-C* converts the expression **"2+3*4"** to the call:

```
_int(2+3*4);
```

and the function **_int** actually displays the value. All of these functions are provided in source form so that you can edit them to satisfy your formatting desires. You can even, if you wish, delete the display statements from the functions, so that no output is displayed. We think, however, that you will find these displays a reassuring confirmation that the system is doing what you asked it to.

5.2 Invoking Functions

Any function in the *Instant-C* libraries or in your program can be called directly from the interpreter. You simply type a normal function call complete with parens and commas, and *Instant-C* will call the function. If the function returns a value (i.e., is not declared **void**), *Instant-C* will print its value as described in the section on expressions above.

The printing of function results is one of the most powerful debugging aids in *Instant-C*. You can easily test any sub-piece of your program by calling the function, and examining the resulting display. You can also easily vary the arguments to the function to see how the value changes.

However, this printing of the resulting value can sometimes cause confusion because you may not think of a particular function as having a value. Since the C language definition requires all functions with no specified data type to be treated as int's, many of the functions you think of as non-valued must be treated as int functions by *Instant-C*. The most frequent surprise is the **main()** function. *Main()* will run your program and print out the last value computed in **main** -- often a meaningless value. This may confuse you since in most other environments **main()** can't return a value. You can change this behavior either by changing the **main** function to be a **void**, or by changing the **_int** function.

Instant-C tries to help in the **main()** case by converting any definitions of **main()** without a specific data type to data type **void** instead of to data type **int**.

5.3 **#run** command

So that you can test the main function of programs that will become separate .EXE or .CMD files, we have provided the **#run** command. For more details, see the description of the **#run** command in Chapter 6.

5.4 Debugging Overview

Instant-C has powerful debugging capabilites that are always available to you. No special libraries, compiler options, or separate utilities are necessary. The debugging facilities offered by *Instant-C* fall into three categories:

Interpretation
: Because *Instant-C* has the interactions of an interpreter, you can perform a number of actions that are difficult or impossible with a compiler. Immediate execution lets you call a function directly to view its value and any side-effects. Arguments can be varied each time you invoke the function to verify proper operation. Variables can be displayed or modified at any time. Execution of your program may be interrupted to allow you to examine its progress or to pursue a different path of execution.

Fast Modification
: Because it is so fast and easy to change your programs, debugging techniques formerly called "brute-force" methods are now elegant and efficient. For example, **printf** calls can be inserted in a function to display the values of certain variables, or to record the

occurance of some events. The function can be tested right away, and once the information is obtained, the debugging code can be removed in a matter of seconds. Thus, *Instant-C*'s debugging capabilities can be extended and customized by your own C language programming.

Debugging Commands
Commands are available to specify functions that are to be traced, to start, continue, or stop execution of your program, and to examine variables and the program execution history.

5.5 Interrupting Your Program

Most debugging activities take place after your program has started running and is interrupted. Interruptions can be voluntary: a call is made to a traced function (see **#trace** command), a call is made to the breakpoint function (i.e., **_()**) function, or a Control-Break interrupt or Control-C interrupt is issued from the keyboard. The keyboard interrupts are available on MS-DOS and PC-DOS systems only. Involuntary interrupts include division by zero, stack overflow, call to an undefined function, or taking the difference of dissimilarly typed pointers. A call to the **_exit** function is considered an interruption so that you can see how your program terminated.

When an interruption occurs, *Instant-C* displays a message describing the interruption. For example, entering

 # i = 3/0

Results in:

```
** Execution interrupted:  division by zero in
command line
```

After the message is displayed, you are at interpreter level. You can issue any command, execute any statement, or (usually) resume execution of your program.

The stack, or history of function calls to the interruption, is preserved, and you can look at it with the **#back** command. At interpreter level, you can display variables, evaluate expressions, and execute C language statements including calls to functions. The C language you enter to the interpreter is evaluated in the context of the function that was interrupted, so you can examine or modify local, or automatic, variables declared in that function.

We use the term "active" to describe functions that are on the stack. The interpreter can execute in the context of any active function by using the **#local** command. **#local** specifies the function to use.

Calling other functions from the interpreter may result in another interruption. Interruptions themselves may be stacked or nested. The **back** command shows the level of any stacked interruptions. The **#reset** command allows you discard or unstack any or all levels. Levels should be discarded when you no longer need them, to prevent confusion and to avoid possible stack overflows.

5.6 Debugger Commands

The debugger commands are fully described in Chapter 6, but an overview of each is offered here so that you can see how they work together.

The commands fall into several categories: those that resume or abort interrupted executions (**#go**, **#step**, and **#reset**), those that manage breakpoints (**#trace** and **#untrace**), and those that display values (**#pc**, **#pd**, **#po**, **#ps**, **#px**, and **#local**).

Any execution that has been interrupted can be resumed in several ways. Resuming execution is much like a executing **return** C language statement from the interpreter. The various ways your can resume your program are:

#go
: resumes execution at the point of interruption, and proceeds until your program returns to the interpreter or another interruption occurs.

#step
: resumes execution, but will breakpoint at the end of the next statement. This is the finest level of control, and allows you step through your program or portions of your program, and observe the flow of control.

#step return
: resumes execution, but will breakpoint when the interrupted function returns. This command is useful in quickly bypassing functions that are not of direct concern to your debugging. For example, you have been single stepping a function that calls **printf**, **step return** will let **printf** execute at 'full speed' until it completes. You can then resume single stepping if appropriate.

#step out
: resumes execution, but will breakpoint when the current function calls another function or returns. This allows you to follow the execution of your program by stepping from function call to function call or return, without line-by-line detail.

#step in
: resumes execution, and like **#step** will break at each statement, but will not notify you of calls to other functions. This makes it easier to use a single command to examine the execution of a function.

#step exec *C_statement*
: unlike the other step commands, **step exec** is a 'call' to the C language statement. This is used to 'step into' a function without having to explicitly

#trace it and then call it. For example,

 # step exec main(argc, argv)

will call **main** and will break immediately upon entry to main.

You can use the following commands to control the debugger in various ways and to display the data and memory of your program:

#reset *level_number*
causes interruptions to be discarded down to *level number*. If no number is specified, zero is assumed and all levels are discarded.

#trace *function_name*
a breakpoint interruption will occur the next time that the specified function is called or returns. If *function name* is omitted, a list of all traced functions is displayed.

#untrace *function_name*
removes the trace request from calls or returns to the specified function. If *function name* is omitted, all trace requests are cancelled.

#pc, **#pd**, **#po**, **#ps**, **#pu**, **#px** *expression count*
display storage from the location of *expression*, (like a C language lvalue), for count locations, under the appropriate printf format (character for **#pc**, decimal for **#pd**, etc.).

#local *function_name*
Sets the context for evaluation of names and expressions at interpreter level. *function name* must be an active function (otherwise references to automatic storage class variables are meaningless). If no function name is given, the name of the current local function is displayed. Upon interruption, the interrupted function

is automatically made the local function.

#back Displays the sequence of function calls to the point of interruption, and also displays any previous interruption levels that have not been **#reset**.

Chapter 6

Instant-C Interpreter Command Reference

This chapter contains a description of each command that the *Instant-C* interpreter recognizes. For each command, we list:

Purpose:	what we intended the command to be used for;
Format:	the syntax of the command;
Remarks:	some explanatory remarks noting major features and possible pitfalls; and
Examples:	some sample uses.

All command names have two spellings: the first is a simple name, and the second is the same name with a leading '#' character. We originally started all interpreter commands with the '#' character so that the command names wouldn't conflict with any names in your programs. After using *Instant-C* for a while, many of our test users felt that they were overcome by '#'s. Having both forms of the names available means that you generally won't need to type the '#'s, but if you want to have a function named "**ed**", you can. The examples and remarks use the two spellings for command names interchangeably.

In the format description for each command, the fixed portion of each command is in **bold type**, while variable parts of the command are in *italics*.

Purpose: To display a back trace of an interrupted execution.

Format: **back**
#back

Remarks: This command displays the functions active on the execution stack at the time execution was interrupted. Also, any prior levels of interrupted execution are displayed.

Backtraces are useful in determining how you got to the point of interruption.

The display begins with a message describing the current execution level, why the execution was interrupted (breakpoint, division-by-zero, entering a traced function, etc.). and the source code that was being executed when the interruption occurred. This is followed by the active functions, listed with the most recently invoked function first. The source code for each active call is shown.

The oldest invocation is always the *Instant-C* command line that started the execution. If more than ten functions are active in the current level, only the first ten are shown, and any others are indicated by an ellipsis (". . .").

Every execution interruption leaves the *Instant-C* interpreter in control, so that you can execute any function or C expression in addition to issuing debugging commands such as **back**. These nested or higher level executions may also be interrupted (by unintended program fault or by request). **#back** will show a summary description for every interruption level, but will display function-by-function detail for only the most recent, or highest, level.

If you type **#back** and no execution interruption has occurred, or the environment has been **#reset**, you will see the message "(no levels active for backtrace)".

See the commands **#reset** and **#go** for more information on the management of interruption levels. See **#local** for information on how to reference local variables in an active function.

Example: # *back*

Displays the current function caller backtrace for the most recent interruption of your programs' execution.

Purpose: To remove an object (function, #define, or data declaration) from the current memory file.

Format: **delete** *name*
#delete *name*

Remarks: If *name* is not in the current memory file, *Instant-C* will print an error.

Name will be removed from the current memory file.

The old definition of *name* will still be available to be listed or edited, until you create a new definition for that name.

Example: # *delete buttercup*

The function *buttercup* is removed from the current memory file. When the memory file is written to disk with the **save** command, the definition of *buttercup* will not be included.

Purpose: To display the filenames in the current disk directory.

Format: **dir**
dir *d:filename.ext*
#dir
#dir *d:filename.ext*

Remarks: *d:filename.ext* is a filename with optional extension and optional disk drive letter. If you don't specify the disk drive, the current disk drive is assumed. If you don't specify an extension, blanks are assumed.

The global characters '?' and '*' may be used in either the filename or the extension, following normal operating system conventions. If you omit the file specifier completely, "*.*" is assumed, and all directory entries will be displayed.

The filenames are displayed five per line on your screen.

Only the filename is displayed; no size or date information is included.

Example: # *dir b:*.c*

The names of all files with an extension of **.C** on the **B:** disk drive are displayed.

Purpose: To switch to the *Instant-C* editor so that you can modify an existing C function or create a new function. You can also use the editor to examine existing functions in order to understand them.

Format: **ed** *name*
ed
#ed *name*
#ed

Remarks: *name* can be the name of a function. (This is the most common use.) *name* can also be the name of a data variable, structure tag, or **#define**'d name. In each of these cases, you will be editing the C source declaration for that item.

If *name* is omitted, the most recently edited object is used. (If *name* is omitted in the very first **ed** command of a session, an error message is given.)

If no object *name* exists, *Instant-C* assumes you are trying to create a new function, and starts with a simple skeleton for the function definition.

One of the most common errors in *Instant-C* is the attempt to edit a file (by specifying a file name) instead of a function or data name. *Instant-C* does not need to compile entire source files from disk, but works directly on individual objects in memory.

Example: *# ed hello*

The function **hello** is loaded into the editor for modification or browsing.

Purpose: To erase specified files from the current disk directory.

Format: **erase** *d:filename.ext*
#erase *d:filename.ext*

Remarks: *d:filename.ext* is a file specifier with optional extension and optional disk drive letter. If you omit the extension, blanks are assumed. If you omit the drive letter, the current default disk drive is used.

The global characters '?' and '*' can be used in either the file name or the extension, following standard operating system convention.

If you specify "*.*" (erase all files), you will be asked for a confirmation that this drastic action is okay.

Example: # *erase oldfile.c*

Erases a file named "oldfile.c" from the current directory on the default disk drive.

Purpose: To resume execution that has been interrupted.

Format: **go**
#go

Remarks: Execution of your program can be interrupted by a fault (e.g., division by zero), or by request (e.g., a breakpoint). The **#go** command will resume an execution that has been interrupted.

Not all executions can be resumed. Examples of interruptions that are not resumable are: division-by-zero fault, missing function fault, call to the function **_exit()**, and stack overflow fault.

If the interrupted execution cannot be resumed, the **#go** command will display an error message, and revert to the previous execution level. This is equivalent to using the **#reset** command to return to the previous level. You can use the **#back** command to examine the execution level to see if you want to resume it, and use the **#reset** command to dispose of level(s) that you don't want to resume.

Example: # *go*

Resumes execution at the point of interruption, or resets the level if it is not resumable.

Purpose: To process a series of interpreter or debugger commands that are stored in a disk file.

Format: **infile** *filename*
#infile *filename*

Remarks: The **infile** command switches input to the interpreter/debugger to come from a disk file.

The most common use of **infile** is to issue all the **load** commands necessary to include all the files of a multiple source file program.

When the end of file is reached, input is switched back to the keyboard.

Under MS-DOS or PC-DOS, you can also use the operating system's command line redirection to execute commands from a disk file

Example: # *infile loadsys.inp*

Reads interpreter commands from the file "LOADSYS.INP". When the end of the file is reached, command input will revert to the console.

Purpose: To display C source language on your screen. You can display either a single function/variable/#define or an entire memory file.

Format: **list**
list *name*
#list
#list *name*

Remarks: The *name* can be either a function name, or a data variable name, or a **#define**'d name. In each case the C source language definition for the named object is displayed on your screen.

If *name* is omitted, the entire current memory file is displayed.

If the named object is not in the current memory file, it must have external scope. If you wish to display a static variable not in the current memory file, you must first switch to the memory file that contains the variable.

To display a different memory file, first switch to that memory file with the **#use** command, and then give a **#list** command with no arguments.

To display C source language to the printer, see the **#llist** command.

Examples:

 # list isdigit

```
int isdigit(c)
char c;
    {
    return c >= '0' && c <= '9';
    }
```

The C source language definition of isdigit is displayed on your screen.

 # list

```
/* C program for "Hello, World!" */

void main()
    {
    printf("Hello, World!\n");
    }
```

(Assuming that HELLO.C is the current memory file.)

Purpose: To list all of the memory files that currently exist in the *Instant-C* workspace.

Format: **listfile**
#listfile

Remarks: All memory files that *Instant-C* has **#loaded** or **#included** in the current session are displayed, one name per line.

You can use the **#use** command to display the name of the current memory file, and to select another file to be the current memory file.

Example: # *listfile*

```
=>  *  (Unnamed memory file)
    lsl
    stdio.h
```

Here, the "*" represents a memory file without a name. It can nonetheless be **#saved** and **#used**.

Purpose: To display all of the names defined or declared in the current memory file.

Format: **listname**
#listname

Remark: Each function definition, data or function declaration, typedef, and #define in the current memory file is displayed, one name per line.

Example: # *listname*

For file hello:
Function main

(Assuming that HELLO.C is the current memory file.)

Purpose: To print C source language on your printer. You can print a single function, variable, or #define, or you can print an entire memory file.

Format: **llist**
llist *name*
#llist
#llist *name*

Remarks: The *name* can be either a function name, or a data variable name, or a **#define**'d name. In each case, the C source language definition for the named object is printed.

If you omit *name*, the entire current memory file is printed.

To display source on your screen, see the **#list** command.

Examples: **#** *llist isdigit*

Sends the C source language definition of the function **isdigit** to your printer.

*llist*

Sends all of the current memory file to your printer.

Purpose: To load a C source language disk file into a memory file.

Format: **load** "*filename*"
#load "*filename*"

Remarks: The **#load** command brings the specified C language source disk file into a corresponding memory file.

If you don't suppoy the extension of the disk file name, it is assumed to be ".C".

Only files in the current disk directory can be loaded.

You can use the characters '<' and '>' instead of double quotes to delimit the file name.

If the filename and extension are each a valid C identifier, you don't need the delimiting characters.

A **#loaded** memory file becomes the current memory file.

Examples: # *load labels*

Reads the disk file LABELS.C into a memory file named "LABELS.C". "LABELS.C" becomes the current memory file.

load <stdio.h>

Reads the disk file STDIO.H from the current disk directory into a memory file named "STDIO.H".

Purpose: To set or query the function assumed for symbol table searches so that you can examine/modify local variables.

Format:

local *function_name*
local
#local *function_name*
#local

Remarks: *Instant-C* normally recognizes names if either they are in the current memory file, or if they are external. The **#local** command lets you evaluate expressions inside a particular function.

The *function_name* specified must be an active function, that is, it must have started executing and have an entry on the execution stack. Expressions that set or use a function's local variable will be executed in the context of that function.

You will generally not need to use the **#local** command because the local function is automatically set from the context of the interrupted execution. For example, if function **f** is executing and attempts a division by zero, the *Instant-C* interpreter is invoked, and the assumed local function is **f**.

If *function_name* is not specified, the current local function name is displayed.

Local function names are stacked with each interruption level; **#resetting** or **#going** to a previous level will result in the previous level's function being re-assumed.

Example: # *local f*

Assume that function **f** calls function **g**, which calls function **h**. Each function has local integer variables named **i**, and **j**. There is a global integer **x**. Execution has been interrupted in function **h** by a breakpoint. The context of function **h** is assumed at the time of the breakpoint, but is changed to function **f** by the example command. Now entering *i = j+x* will set **i** in **f** to the value of **j** in **f** plus the global **x**.

Resuming execution with *#go*, **h** returns to **g**, which returns to **f**. Function **f** continues execution with the new value in its local variable **i**.

Purpose: To create a stand-alone version of your program after it has been debugged.

Format:

make *filename*
make *filename starting_function*
#make *filename*
#make *filename starting_function*

Remarks: You must supply *filename*.

You use **make** to create separate programs from *Instant-C* to be run under your operating system. None of the special debugging features such as break checking are available, since the interpreter is not written to the file.

Make will overwrite existing files.

If you omit the extension portion of *filename*, it defaults to **.EXE** under MS-DOS and PC-DOS, and to **.CMD** under CP/M-86.

Because the created module contains the entire library, it will be relatively big (>32KB). If you don't need the entire library for your program, you can use ICBASE to **load** and **make** your program. If you use ICBASE, the minimum size is about 3KB.

Make is saving an exact memory image of your program and data values. Note that some data values may be invalid when the created module is executed if you have obtained absolute paragraph memory addresses.

Test and debugging runs may allocate storage that is **not** reclaimed -- **make** will save all memory that has been used by your program during the entire *Instant-C* session.

You will normally omit the *starting_function* name, in which case the **_main** function in the library will be used. **_main** parses the command line and calls your **main** function with the standard **argc, argv** arguments.

Examples: # *make prtlabel*

Writes the disk file "PRTLABEL.EXE" ("PRTLABEL.CMD" under CP/M-86). All currently loaded functions and data are included. When the PRTLABEL file is executed, the library function **_main** will call your **main** function with the correct **argc**, **argv** arguments.

make setwide init

Writes the disk file "SETWIDE.EXE" (or "SETWIDE.CMD"). Execution will begin with the **init** function. (The **_main** function will not be called.)

Purpose: To delete all objects from the current memory file, or to create a new memory file.

Format:

new
new *filename*
#new
#new *filename*

Remarks: If you omit the *filename*, the current memory file is reset. Resetting a memory file means deleting all of the objects declared or defined in it.

If you provide the *filename* and there already exists such a memory file, that memory file becomes the current memory file and is reset to be empty.

If you provide the *filename* and no such memory file exists yet, a new, initially empty memory file is created and becomes the current memory file.

Examples:

\# *new*

Deletes all names in the current memory file.

\# *new part2*

Creates a new memory file named "part2" and makes it the current memory file.

\# *new **

Clears the unnamed memory file (after possibly creating it).

Purpose: To redirect interpreter output to different and/or multiple devices.

Format: **outfile printer**
outfile crt
outfile both
#outfile printer

Remarks: **outfile printer** directs output to your printer and not to your screen.

outfile crt directs output to your screen and undoes a prior **outfile printer** or **outfile both.**

outfile both directs output to both your screen and your printer.

Only the output of the interpreter is redirected. Neither the output of the editor, nor any output of your program is affected in any way.

Even if output is directed to your printer only, any error messages will also be displayed on your screen.

Example:

```
# outfile printer
# list main
# outfile crt
```

Makes a listing of the **main** function on your printer. (The **#llist** command could be used to do the same thing.)

```
# outfile printer
# back
# outfile crt
```

Prints a backtrace of your currently interrupted program on the printer.

Purpose: To display memory locations in character format.

Format: **pc** *expression count*
#pc *expression count*

Remarks: *expression* is evaluated as an lvalue (left-hand part of an assignment), and bytes beginning at that location are displayed as characters. If the optional *count* expression is included with the **#pc** command, then *count* bytes are displayed as characters. If *count* is omitted, only 1 byte is displayed.

Any non-printing values are displayed as '?'.

The library function **_pc** is called by the *Instant-C* interpreter to implement the **#pc** command. You can alter the display format or actions by changing the function **_pc**.

See the commands **pd, po, ps** and **px** to display data in other formats.

Examples:

*# pc *str 10*

Assuming the declaration **char str[10]**, the first ten characters of **str** are displayed as characters.

 # pc str[0] 10

This will have the same results as the first example.

 # pc i

The variable **i** is displayed in character format. Although i (declared as **int i;**) is two bytes in storage, only one byte is displayed.

 # pc 0x2174 0x10

This will display 16 bytes as characters starting at location 2174 hex in your program's data. (0x10 is 16 decimal.)

Purpose: To display memory locations in decimal format.

Format: **pd** *expression count*
#pd *expression count*

Remarks: *expression* is evaluated as an lvalue (left-hand part of an assignment), and words beginning at that location are displayed as decimal integers. If the optional *count* expression is included with the **#pd** command, then *count* words are displayed as decimal integers. If *count* is omitted, only one word is displayed.

The library function **_pd** is called by the *Instant-C* interpreter to implement the **#pd** command. You can alter the display format or actions by changing the function **_pd**.

See the commands **pc**, **po**, **ps**, and **px** to display data in other formats.

Examples: *# pd *ia 10*

Assuming the declaration **int ia[20]**, the first ten words of **ia** are displayed as decimal integers.

 # pd i

The variable **i** is displayed as a decimal integer.

Purpose: To display memory locations in octal format.

Format: **po** *expression count*
#po *expression count*

Remarks: *expression* is evaluated as an lvalue (left-hand part of an assignment), and words beginning at that location are displayed as octal integers. If the optional *count* expression is included with the **#po** command, then *count* words are displayed as octal integers. If *count* is omitted, only one word is displayed.

The library function **_po** is called by the *Instant-C* interpreter to implement the **#po** command. You can alter the display format or actions by changing the function **_po**.

Examples: # *po *ua 10*

Assuming the declaration **int ua[35]**, the first ten words of **ua** are displayed as octal integers.

po ua[27] 7

Displays the last seven elements of **ua** in octal, beginning with element 27.

po i

The variable **i** is displayed as an octal integer.

Purpose: To display memory locations as character strings.

Format: **ps** *expression count*
#ps *expression count*

Remarks: *expression* is evaluated as an lvalue (left-hand part of an assignment), and pointers starting at that location are displayed as character strings. If the optional *count* expression is included with the **#ps** command, then *count* pointers are displayed as character strings. If *count* is omitted, only one pointer is displayed.

The library function **_ps** is called by the *Instant-C* interpreter to implement the **#ps** command. You may alter the display format or actions by changing the function **_ps**.

Any non-printing values are displayed as '?'. A string is assumed to continue until a byte with value 0 is found. (At least as implemented by the **_ps** function.)

See the commands **pc**, **pd**, **po**, and **px** to display data in other formats.

Examples: # *ps answer*

Assuming the declaration **char answer[100]**, the characters beginning at **answer[0]** are displayed until a byte with value 0 is found, indicating the end of the string.

ps answers[0] 10

Assuming the declaration **char *answers[10]**, an array of character string pointers, this command will print each character string.

ps i

The variable **i** is interpreted as a character string pointer, and data at ***i** is displayed as a character string.

Purpose: To display memory locations in hex format.

Format: **px** *expression count*
#px *expression count*

Remarks: *expression* is evaluated as an lvalue (left-hand part of an assignment), and words starting at that location are displayed as hex integers. If the optional *count* expression is included with the **#px** command, then *count* words are displayed as hex integers. If *count* is omitted, only one word is displayed.

The library function **_px** is called by the *Instant-C* interpreter to implement the **#px** command. You can alter the display format or actions by changing the function **_px**.

See the commands **pc**, **pd**, **po**, and **ps** to display data in other formats.

Examples: # *px *ia 10*

Assuming the declaration **int ia[20]**, the first ten words of **ia** are displayed as hex integers.

px i

The variable **i** is displayed as a hex integer.

px 0x2174 16

This will display 16 words starting at location 2174 hex in your program's data area.

Purpose: To return to the operating system.

Format: **quit**
#quit

Remarks: You use the **quit** command when you are finished using *Instant-C*.

No memory files are automatically saved to disk If you have changed your program, make sure you have updated it to disk with the **save** command, or with a **#savemod** command.

You can also use the **system** command to do the same thing.

Example: # *quit*

Returns you to the operating system. Any modifications you have made to your programs and have not saved are lost.

Purpose: To change the name of a function, variable, or a **#define**d symbol.

Format: **#rename** *oldname newname*
#rename *oldname* **as** *newname*

Remarks: The **#rename** command changes the name of the object, and all of the references to the object. This makes one part of software maintenance much simpler.

All references in the *Instant-C* workspace are changed. No references in any disk files are modified, unless you **save** the workspace memory files to disk. Similarly, no references in comments or character string literals are modified.

#rename will result in an error message if *oldname* doesn't exist already, or if *newname* exists already.

Since there is already a **rename** function in the standard *Instant-C* library, you must provide the leading # in the **#rename** command.

Example: # *#rename islower issmall*

Changes the name of the library function **islower** to be **issmall.** All references to this function, (including, for example, the reference in the library function **toupper**) are changed also.

Purpose: To discard one or more nested interrupted execution levels.

Format: **#reset**
#reset *levelnumber*
reset
reset *levelnumber*

Remarks: The **#reset** command allows you to get rid of an interrupted execution level when it is no longer needed. You will not be able to resume execution with the #go command nor will the level be available for examination of its stack history).

#reset will revert the execution level to that indicated by the optional *levelnumber* parameter, or to level 0 (no levels active) if *levelnumber* is not provided.

If *levelnumber* is higher or equal to the current level number, no action is taken.

Use the **#back** command to display the levels that are active; you may use the **#go** command to resume execution of the current (highest) level.

Whenever your program is interrupted (by fault such as division-by-zero or stack overflow, or by request, such as a breakpoint), a new interrupted execution level is created.

Example: # *reset 2*

This command will throw away any level information for levels higher than 2.

Purpose: To execute your program as though it were invoked from the operating system.

Format: **run** *command_arguments*
run
#run *command_arguments*
#run

Remarks: You can test stand-alone programs with the **run** command. Your program executes as it would if it were started by the operating system.

Instant-C invokes your program by calling **main** with the normal **(argc, argv)** convention of passing arguments.

The *command_arguments* are separated by using space characters as delimiters.

argv[0] will contain the string **"main"**.

If no *command_arguments* are given, **argc** will be 1.

The **run** command actually calls the library function **_main** to parse the command line and to call your **main** function. **List** the **_main** function for more details.

Examples: (Assumes HELLO.C as the current memory file.)

run
Hello, World!

run this line can be 128 chars long
Hello, World!

The output is the same since the HELLO.C program ignores its arguments.

Purpose: To write the current memory file to disk, thereby saving any updates, additions or changes that you have made to your programs.

Format:

save *new_filename*
save
#save *new_filename*
#save

Remarks: The name of the current memory file is used for the created file, unless you provide a *new_filename*.

The file extension defaults to ".C" if you don't specify one.

If the file already exists, the existing file is renamed to have file name extension ".BAK". If you discover that you made a mistake, you can use the ".BAK" file to get the most recent previous copy back.

See the **#use** command to display/change the current memory file name.

Examples:

```
# use
    HELLO.C
# save
#
```

Writes the current memory file to the disk file "HELLO.C". If "HELLO.C" already exists, it will be renamed to "HELLO.BAK".

```
# save hello.new
#
```

Writes the current memory file to the disk file HELLO.NEW.

Purpose: To save a new or updated version of *Instant-C* to disk in binary form, including all symbol tables and system options.

Format: **savemod** *filename*
#savemod *filename*

Remarks: You must supply *filename*.

Savemod will overwrite existing files, so be careful not to overwrite an existing file unless you are sure everything will be okay.

If you omit the extension portion of *filename*, it defaults to **.EXE** under MS-DOS and PC-DOS, and to **.CMD** under CP/M-86.

You can use **savemod** to change *Instant-C*'s defaults, to update the built-in libraries or to add your own library functions.

You can also use **savemod** to save a partially debugged program when you need to stop working before your program is finished. In this case, you are adding your programs and workspace to a clone of *Instant-C*.

The created modules will be very large (> 210K), so be sure you have sufficient disk space.

Savemod saves an exact memory image of your program and data values. Note that some data values may be invalid when you start up the saved version, such as absolute memory addresses you may have developed.

Savemod may not be given if there are any interrupted executions of your program pending. (You can tell that nothing is pending if a **#back** command responds with "no levels active".) The **#reset** command will remove any pending executions for you.

Examples: # *savemod newic*

Writes NEWIC.EXE (or NEWIC.CMD) to disk. After testing NEWIC, you can replace the existing *Instant-C* with NEWIC.

savemod sortbug

Writes SORTBUG.EXE (or SORTBUG.CMD) to disk so that you can restart it later and continue your development or debugging.

Purpose: To display the paragraph address, maximum size, and currently used space in each of the memory segments used by *Instant-C*.

Format: **segments**
#segments

Remarks: All data is displayed in hexadecimal.

You will probably never need to use this command. We provided it to help the debugging of **Instant-C** itself.

Your program is stored in the segments labeled "User Code", "User Data", "Symbol" and "Source". By looking at these values before and after loading a file, you can tell how big your programs are.

Example: # *segments*

Displays the segment use information for *Instant-C* on your screen.

Purpose: To temporarily switch to the operating system to execute a few commands or to execute a single operating system command.

Format:

shell
shell *command line*
#shell
#shell *command line*

Notes: The **shell** command is only available under MS-DOS or PC-DOS and requires version 2.00 or later of those operating systems.

The **shell** command requires extra memory space beyond that of *Instant-C*. As a result, you will only be able to use it if you have at least 512KB memory on your system.

If you provide a *command line*, that single command is executed. If you omit the *command line* argument, you are transferred to a new copy of COMMAND.COM. You can get back to *Instant-C* by giving the DOS **exit** command.

Example:

\# *shell*

Invokes a new copy of the DOS command processor. You can get back by typing "exit".

\# *shell cd*

Executes the "*cd*" command to query the current directory.

Purpose: To resume execution and interrupt after the next statement is executed.

Format: **step**
#step

Notes: You use the **#step** command to execute a single statement in the interrupted function.

#step will stop after the next statement, whether it is in the current function, a function called from the current function, or the function which called the current function.

See the **#step exec**, **#step in**, **#step out**, and **#step return** commands for other ways to resume and control execution of your programs.

Example: # *step*

Interrupts your program after execution of one more statement.

Purpose: To execute a C statement or function call and to interrupt after the first statement.

Format: **step exec** *C_statement*
#step exec *C_statement*

Notes: The *C_statement* will most often be a single function call.

The **step exec** command provides a simple alternative for functions which are called only once to the **trace** and **untrace** commands.

See the **#step**, **#step in**, **#step out** and **#step return** commands for other ways to resume and control execution of your programs.

Example: *# step exec fp = fopen("PRN:", "w")*

Executes a call to **fopen** and will interrupt execution at the beginning of the **fopen** function.

Purpose: To resume execution and to interrupt after the next statement in the currently active function.

Format: **step in**
#step in

Notes: You use the **step in** command when you want to breakpoint in the same function, and not in any lower-level functions.

The **step in** command allows you to stay at one level in a function heirarchy.

See the **#step**, **#step exec**, **#step out** and **#step return** commands for other ways to resume and control execution of your programs.

Example: # *step in*

Executes any nested calls without interruption, and stops after the next statement in the currently interrupted function.

Purpose: To resume execution and to interrupt after the next statement executed that is not in the currently active function.

Format: **step out**
#step out

Notes: You use the **step out** command to see what happens next outside of the currently executing function.

The execution will be interrupted when the current function either calls another function, or returns to its caller.

See the **#step**, **#step exec**, **#step in**, and **#step return** commands for other ways to resume and control execution of your programs.

Example: # *step out*

Resumes execution and interrupts when the current function calls another one or when it returns.

Purpose: To resume execution and to interrupt when the currently active function returns.

Format: **step return**
#step return

Notes: You use the **step return** command when you are not interested in watching the details of executing the rest of the current function.

Execution will be interrupted when the current function returns to its caller.

See the **#step**, **#step exec**, **#step in**, and **#step out** commands for other ways to resume and control execution of your programs.

Example: # *step return*

Executes the rest of the current function without stopping and stops when the current function returns to its caller.

Purpose: To return to the operating system.

Format: **system**
#system

Remarks: You use the **system** command when you are finished using *Instant-C*.

No memory files are automatically saved to disk. If you have made changes to your programs that you want to keep, make sure you have saved them with the **save** command, or with a **#savemod** command.

You can also use the **quit** command to do the same thing.

Example: # *system*

Returns you to the operating system. Any modifications you have made to your programs and have not saved are lost.

Purpose: To turn on call/return tracing for a function.

Format: **trace** *functionname*
#trace *functionname*

Remarks: The **#trace** command will mark the specified function to issue a breakpoint interruption both when it is called and when it returns. This allows you to watch one or more traced functions to see how and when it/they are called.

You can give a **back** command to see how the traced function was called.

At interpreter level, you can use the normal command line expression evaluation features of *Instant-C* to examine or modify the arguments to the traced function, or to pursue any other path of execution or debugging. Use **go** to begin execution of the traced function, or use **#reset** to abort the execution.

A traced function that interrupted upon entry will also interrupt upon return. The message notifying you of the return displays the return value (in decimal). The function's local variables are available for examination or modification at this time.

When you no longer need a function to be traced, use the **#untrace** command.

Example: # *trace fopen*

Will breakpoint upon each subsequent entry and return from the function **fopen**.

Purpose: To display a disk file in the current disk directory.

Format: **type** *d:filename.ext*
#type *d:filename.ext*

Remarks: If you omit the drive letter, the current disk drive is used.

If you omit the extension, blanks are used.

The disk file *d:filename.ext* is displayed on your screen.

Any tabs are converted to spaces controlled by the built-in system variable **_tabwidth.**

Example: # *type newprog.c*

Displays on your screen the file NEWPROG.C from the current directory on the default drive.

Purpose: To turn off call/return tracing for a function.

Format: **#untrace** *function_name*
untrace *function_name*

Remarks: Undoes the **#trace** command.

There will no longer be an interruption when the function *function_name* is called or when it returns.

Example: # *untrace fopen*

fopen is no longer traced. It will no longer interrupt when called or when returning.

Purpose: To make another memory file within the *Instant-C* workspace be the current memory file, or to display the name of the current memory file.

Format:
use
use *filename*
#use
#use *filename*

Remarks: The current memory file is where all new declarations are put.

If you omit *filename*, the name of the current memory file is displayed.

If your *filename* doesn't include an extension, ".C" is used for default.

If no memory file named *filename* exists, an error message is printed.

You can use the **listfile** command to display what memory files exist.

Examples:

use

HELLO.C

Displays the current memory file name.

use stdio.h

Switches to the memory file named "stdio.h" for edits and new declarations. ("stdio.h" must have been previously created or loaded.)

Chapter 7

Using Instant-ED on Files

Instant-ED, the editor in Instant-C can be used outside of the Instant-C environment. This chapter covers how to use *Instant-ED* in stand-alone mode.

7.1 Command Line Syntax

The *Instant-ED* editor can be used to edit any disk file that fits within its space limitations. It is invoked from your operating system's command level.

The syntax of the *Instant-ED* command is:

ed [*filename*] [*options*]

where both the *filename* and the *options* may be omitted. If a *filename* is given, and the disk file exists already, it read into the editor's buffer before the editor will start acting on your keystrokes.

If a *filename* is given, and the disk file doesn't yet exist, *Instant-ED* creates a new disk file when you give a **F** or **W** command.

The possible *options* are:

Option	Example/Meaning	
+<tab setting>	+8	for tabs every eight columns
-<initial line>	-123	for initial cursor at line 123.

The default tab setting is every eight characters for disk files whose extension starts with "A" (.ASM, .A86, etc.), and every four characters otherwise.

The initial cursor line option is particularly useful with traditional compilers when you are given a line number for some error message. You can start up **ED** on the line where the traditional compiler found the error.

The editor operates on disk files in stand-alone mode in essentially the same way as it does on functions when you are inside Instant-C. Most of the descriptions of the editor that are in the following chapter "Instant-ED Reference" apply identically when operating on disk files in stand-alone mode.

The major difference is that no compilation automatically occurs when exiting the stand-alone version of the editor. The following describes the two ways of exiting from the editor:

Ctrl Q Exit without saving text contents to a disk file.

Ctrl F Write the text contents to disk and exit if successful. Saving the contents when not in Instant-C does not imply successful compilation of the text.

Chapter 8

Instant-ED Reference

This chapter covers the use of the editor in *Instant-C*, including the examination and change of functions and other declarations in your programs.

8.1 Starting the Editor

Enter

ed object_name

where **object_name** is optional. **object_name** is the name of any function, data declaration, or **#defined** symbol. If you do not supply a name to be edited, *Instant-C* assumes that you want to edit the last function or declaration that you were editing. If you don't supply a name, and if this is the first **#ed** command in your session, *Instant-C* prints an error message. If **object_name** is not declared, *Instant-C* assumes it is a function and will start the editor with a function template that you can fill in with C statements.

8.2 Editor Terminology

These terms will be used throughout the editor chapter. To help get started with the editor, you should know:

buffer — consists of your text, ranging from none to thousands of lines of text. Some editor operations work on the entire buffer, such as the Write buffer to disk file command. There can be more than one buffer, and the editor has functions to move blocks of text back and forth between buffers.

cursor — usually means both the cursor on your console screen and the location in the editor buffer at which text can be inserted or deleted.

command — causes the editor perform some operation. Commands are distinct from editor **functions** in that commands require more information than can be provided in a single keystroke. An example is to Read a file from disk into the current buffer, where you will need to specify the file name.

command argument
additional information needed for a command to execute. For disk file Read or Write, the command argument is a file name. For text change, the command arguments are the text to be replaced and the replacing text.

command line
the top line of the screen. Commands, and their arguments, will be displayed here. Also, messages from *Instant-C* or error messages from the editor will be displayed here.

function or **key function**
a simple editor operation, which requires no additional information to complete. In general, you can cause these functions to operate by a single keystroke. Examples of functions are: move cursor down one line, delete character to the left of the cursor, change to command mode.

status line — the second line of the screen. Always displays the editor mode, buffer name and type, and cursor location.

8.3 Display Layout

Commands to and messages from the editor are displayed on the first, or **command**, line at the top of the screen. See the "Command Mode" section below for details on what the commands are, and how you make them happen.

The current editor mode is always indicated on the second, or **status** line, along with the current buffer (indicated by the type and name of the buffer), the line number of the cursor within the buffer, and the column number of the cursor.

All other lines of the screen are used to display the text in the current buffer. The screen will always show the text area surrounding the current cursor position.

8.4 Editor Modes

ED can be in one of three modes, which determine the editor's response to typing from the keyboard. The three modes are Insert (the default and most used), Overtype, and Command.

While in Insert mode and Overtype mode all printable characters typed will appear at the cursor on the screen. Insert mode creates new text characters in the buffer and on the screen, while Overtype mode replaces existing characters with the new characters.

Most non-printing characters or function keys may cause other actions to occur, such as cursor movement, mode change, deletion of characters or lines, etc. These are called "key functions"; most key functions can be used at any time, whether in Command, Insert or Overtype mode.

Command mode is an escape from this direct typing, and allows the use of printable characters to instruct the editor to perform other functions, such as changing the next occurrance of a string of characters to be a new string, going to a specific line in the buffer, or quitting the editing session.

You can, if you wish, define the keystrokes which will cause any function to occur. (See "Keyboard Configuration" in Appendix E.) You can change the definitions to match another editor with which you are familiar, to be easy for your own typing style, or to take advantage of special labels on the keys of your keyboard.

8.5 Editor Input Modes (INSERT and OVERTYPE)

In the input modes, certain characters input from the keyboard will cause editor functions to occur (such as the deletion of a word, or movement of the cursor). Any character not defined to cause a function will be placed into the buffer, and will be displayed on the screen. In Insert mode, the character typed will be inserted into the current line of text; all other characters on that line, and the cursor, will move to the right by one character position. In Overtype mode, the character at the cursor is replaced by whatever new character is entered, and the cursor moves to the right by one position.

In either mode, the cursor can move to columns greater than the width of the screen ('past' the right side of the screen, in other words). When this happens, the text on the screen is shifted to the left so that the cursor can remain on the screen (and on the same line as before). The cursor can't go past column 240, though. This feature is called 'horizontal scrolling'. When horizontal scrolling is in effect, the number of columns shifted (the number of columns in the buffer to the left of the first column displayed on the screen) is shown beside the column indication on the status line. "col 82- 4", for example, would indicate that the display has been shifted 4 columns to the right, and the current buffer column position is 82.

Non-printable control characters that you Insert or Overtype are handled in a special way. This is to prevent confusion on input (many control characters typed on the keyboard will be interpreted as key functions), and to allow display on the screen. Certain control characters, such as tab (control-I) and carriage return (control-M) are not treated specially, but instead are directly represented on the screen. Tab is represented by spaces up to the next tab column; carriage return appears as the end of characters on a line and continuation on the next.

The caret (or uparrow) key '^' is used to translate the following character to a control character. For example, entering '^x' will place control-X into the buffer at the current location. The caret key also acts as an escape to prevent the interpretation of the next character as a key function. Entering caret and then control-X places a control-X character into the buffer without switching you to Command mode. It is necessary to enter the caret key twice to get a single caret character into the text buffer.

Control characters are displayed with a caret character preceding to indicate that they are control characters (e.g., ^A is control-A). Please be aware of the ambiguity of ^^ in that the character control-caret is displayed the same as the simple caret character. Also, it is better practice to use the backslash escape of the C language (i.e., '\t' or '\032') to represent control characters in source code. Control-@ ('\0') cannot be represented in the buffer.

8.6 Editor Command Mode

Commands may be given either directly with a single keystroke, or as a two-step process where you first enter Command mode, and then select the particular command that you wish to execute. As delivered, the *Instant-C* Editor uses the two-step process because it is more general and easier to learn. (See "Keyboard Configuration" in Appendix E for details on how to custom-configure single keystroke commands.)

Control-X or **F10** will enter Command mode. Another character is needed to select the particular command; in each case the first character of the command name is used. This character will cause an acknowledgement and prompt to appear in the command line. The default value, if any, will appear already filled in after the prompt. Carriage return will start execution with the default. To use any other value, enter it, followed by carriage return. An underscore is used as a cursor on the command line when argument values are being entered. Errors in commands usually require the space bar to be pressed to acknowledge the error and proceed. Control-C cancels a command you have not yet executed. Use the Backspace key to correct the argument value as you type it in.

Control-R is used to repeat the last command. This makes it very easy to browse through a text buffer with the Search command, or to make a global change. Either RETURN or Control-R signals that a command argument is completely entered, and execution of the command may commence.

8.7 Editor Command Summary

Once in command mode, you enter only the first letter of the command name; no return is needed. The editor will respond with the full command name, and prompts for the command arguments. These prompts appear on the command line (the topmost line) of the screen. When arguments are expected to a command, you need to enter a return to mark the end of the argument(s).

Finish	Compile and save the text in current buffer; return to *Instant-C* interpreter when no errors are detected. (Control-F does this also)
Quit	Discard buffer; return to the interpreter without compilation. (Control-Q does this also)

Search *target_string*
Change *target_string replacement_string*

Target_string is a string of characters to search for. The Search and Change commands will look for an exact match in the buffer. Search places the cursor at the end of the *target_string* in the buffer. Change replaces the *target_string* with *replacement_string*. An optional + or - may be entered before **C** or **S** to indicate a search direction: + for forward in the buffer, - for backwards in the buffer from the cursor location. The direction is remembered for repetitions of the command.

Insert *string*

The command argument *string* will be inserted at the cursor.

Line *absolute_line_number*
Line + *relative_lines*
Line - *relative_lines*

Move cursor to the beginning of *absolute_line_number*, or move to the line forward or backward *relative_lines* from the current line.

Read *d:filename.ext*
Write *d:filename.ext*

The current buffer is written to disk, or a disk file is read into the buffer at the cursor. *d:* is an optional disk drive specifier; *.ext* is the filename extension.

Edit *d:filename.ext*

The current buffer is cleared, and the disk file is loaded into the current buffer. This is the same as quitting the editor and restarting with a new disk file, except that the contents of the temporary buffer is not cleared -- thus giving you a way to transfer text between files. This command is available only in the stand-alone version of the editor.

Buffer

select one of these subcommands:

Switch *buffer_name*
switch to another text buffer
Create *buffer_name*
create buffer with given name
Delete *buffer_name*
delete the named buffer

? repeat the editor startup message, which may be a compiler error message with line number and description.

Ctrl-X or **F10**	Enter command mode -- next key will be the command.
Ctrl-C	At any point in Command mode, will abort the current command and return to inser
Backspace	In command mode, erase the last character of the command argument.
Return	The command argument is entered, begin execution.
Ctrl-R	The command argument is entered, begin executing the command. If not in Command mode, **Ctrl-R** will re-execute the last command.

Example of using a command (spaces are shown, but should not be typed):

F10 - s main() Return

i.e., enter function key F10, followed by -, then by *s*, followed by the string *main()*, followed by the carriage return key. This command will search backwards from the cursor for the string "main()".

8.8 Key Functions for the IBM-PC

These functions occur instantly when the proper key or sequence of keys is hit. Not all functions are allowed when ED is expecting a command value. (E.g., one cannot switch to Insert mode while entering a search target string, but one can issue cursor movement function). The key sequences listed here are for the IBM PC and compatible computers; they can be customized to your terminal or your whims (see the section called "Configuring the Keyboard" in Appendix E). With each description is the function number [in brackets] as used by the keyboard configuration program.

8.8.1 Moving the Cursor

Home	[3] cursor to top of buffer
End	[4] cursor to end of buffer
left arrow	[6] cursor left (or to end of previous line if at beginning of current line)
right arrow	[7] cursor right (or to beginning of next line if at end of current line)
up arrow	[9] cursor up vertically, to same column in previous line
down arrow	[12] cursor down vertically, to same column in following line
Ctrl-PgUp	[20] cursor up one line (to beginning of current line, or if there already, to previous line)
Ctrl-PgDn	[11] cursor down (to beginning of next line, or end of current line if end of buffer)
PgUp	[16(14)] 'page' up (move cursor by multiple lines)
PgDn	[18(14)] 'page' down
Ctrl right arrow	[14] cursor word right
Ctrl left arrow	[13] cursor word left

8.8.2 Deleting Characters

Del	[22] delete character at cursor
Backspace	[43(23)] delete character preceding cursor. (In command mode, this key deletes the last character typed in a command argument.)

F6 [24] delete entire line

Ctrl End [25] delete line from cursor to end (same as delete entire line if cursor is at the beginning of the line).

F8 [38] delete word right

F7 [37] delete word left

8.8.3 Inserting Characters

F5 [27] insert new line before current line

Ctrl-Return [10] insert new line following current line, with same indentation as current line

8.8.4 Changing Editor Modes

F10 or **Ctrl X**
[46] enter command mode, next character is command

Ins [39] toggle mode (if in Insert mode, set to overtype mode; if in Overtype mode, set to Insert)

8.8.5 Moving Text

F4 or **Alt F6**
[35] un-delete item (as characters or lines are deleted, the most recent are retained in a 'garbage stack', and can be recalled into the buffer at the cursor location)

F1 [31] set tag to current cursor location. The 'block' of text between the tag and the cursor (as it is moved around) can be treated in special ways. See Appendix F.4, Text Blocks Management, for more discussion.

Alt F1 [32] swap tag and cursor (to see where tag is, or to go back to a saved place)

F2 [33] save block of text between tag and cursor in TEMP buffer

F3 [34] recall TEMP buffer at cursor

8.8.6 Other Editor Key Functions

Ctrl R [47] Initiate command, or repeat last command.

Ctrl C [42] If in Command mode, return to input mode. Otherwise, reset editor and redraw screen.

F9 [26] swap the two characters preceding cursor (on current line only)

Alt F9 [36] swap case of character at cursor (convert upper case to lower, lower to upper)

8.8.7 Leaving the Editor

Ctrl Q [49] Exit without compiling and saving text contents. See "Quit" in the Command Summary above.

Ctrl F [48] Save text and exit if successful. Saving the contents when in *Instant-C* implies successful compilation of the text. If there are compilation errors, you will be put into the editor's INSERT MODE with the error message displayed in the command line, and with the cursor positioned at the point of the error. See "Finish" in the Command Summary above.

Chapter 9

Instant-C Function Library

The *Instant-C* function library contains all of the library functions described in Kernighan & Ritchie that apply to the CP/M-86 and MS-DOS operating systems and their derivatives.

C language source code for these functions is delivered with *Instant-C*, and the source code can be considered the most detailed documentation. Thus, you may modify or extend the function library, but beware! such modification can lead to trouble later when documentation fails to match the actual code, or when dependencies on specific programs become buried in the libraries. Good programs will make the minimum number of assumptions about exactly how a library function does its job.

Some extensions to the 'standard' libraries are also included. These may be either specific to *Instant-C*, or they may be commonly found and expected in C libraries, although not described in Kernighan & Ritchie.

The commonly available functions not defined in K&R are:

```
cgets, cputs, movmem, setmem,
rename, inportb, outportb
```

The *Instant-C* specific functions are:

```
bdos, bdosw, _firstarg, _lastarg,
_int, _char, _string, _ptr,
_dc, _dd, _do, _ds, _dx,
_call, _movdat, _interrupt, _flags, _segread,
```

Various internal functions and variables are declared which should be referenced only by the library functions. These follow the naming convention of an "_" prefix, such as **_printf**. You should not use these items directly, as they may change in definition or may not appear at all in later versions.

The **bdos** and **bdosw** functions perform a call to the operating system, and differ only in the type of the return value (char and int, respectively). **_firstarg** and **_lastarg** are used to support the passing of variable numbers of arguments to library routines such as **printf** and **scanf**. *Instant-C* otherwise performs strict checking as to number of arguments and function return type.

9.1 Library Categories

The library consists of several C-language source files. Each file contains the functions for a particular category of functions, e.g., memory management, IO, etc. These files have the suffix .IC to help distinguish them from other source files.

The *Instant-C* library functions, by category, are listed below.

Source file CTYPE.IC: Character group tests and conversions

isalnum	isalpha
isascii	iscntrl
isdigit	islower
isprint	ispunct
isspace	isupper
isxdigit	tolower
toupper	

Source file STRLIB.IC: String manipulation

strcat	strcmp
strcpy	strlen

Source file MEMORY.IC: Memory management

```
sbrk        getmem
alloc       malloc
calloc      retmem
free        movmem
setmem
```

Source file STDIO.IC: IO functions

```
getc        getchar
getch       putc
putchar     putch
ungetc      fgets
gets        fputs
puts        read
write       create
fopen       fclose
fileno      ferror
feof        clrerr
open        close
lseek       unlink
exit        _exit
cgets       cputs
rename
```

Source file PRINTF.IC: Formatted string IO

```
printf      sprintf
fprintf     scanf
fscanf      sscanf
```

Source file FUNCVAL.IC: Interpreter and Debugger routines

```
_int        _char
_unsigned   _short
_long       _double
_ptr        _string
_pc         _pd
_po         _ps
_px         _main
```

[Note these functions are unique to *Instant-C*. The first functions (_int, ... _string) display the value resulting from any C-language code typed to the *Instant-C* interpreter. These are executed only if they are loaded into *Instant-C* and have not been renamed. (The IC program is delivered to you with these functions pre-loaded.) The others (_pc, _pd, etc.) are called by the corresponding debugger display commands (**pc, pd**, etc.), if the functions are loaded into *Instant-C*. They are supplied as part of the library to allow customization.]

Functions to provide low level services (built-in; no source file)

bdos	bdosw
inportb	outportb
_flags	_interrupt
_movdat	_segread
_call	

[Note these functions are provided as an alternative to some kind of assembly-language interface. These functions provide direct access to hardware resources. They are similar to functions in other C compiler packages, though not necessarily exactly compatible. Any reference to these routines should be in your machine- and system-dependent code sections only.]

Source file INTLIB.IC: Additional system interrupt management functions

interrupt_get
interrupt_set
prologue_init
interrupt_install

[Note: INTLIB.IC functions are not built-in to the IC program as are the other library functions. The source file is provided should you have applications requiring the signalling or handling of hardware interrupts.]

Instant-C standard library header file (STDIO.H).

You can #include STDIO.H in your programs to provide certain frequently used #define's, such as EOF and NULL, and also to provide a definition of the structure for the FILE type. Compatability note: in many C implementations, some functions are implemented as macros (examples include isupper, islower, getc). Because of restrictions on preprocessor support in *Instant-C* version 1, macros are not available. All functions provided by the *Instant-C* library are implemented as functions and not as macros.

9.2 *Instant-C* Library Functions Description

For these brief (and temporary) explanations of the *Instant-C* library, the following short-hand is used to describe the calling sequence (number and type of arguments) of each function, and the returned value, if any. The presumed declarations are:

```
int b;           /* boolean value, true or false */
char c;          /* character value, one byte */
char *cp, *cp1, *cp2;/* character string pointer */
int i, i1, i2;   /* integer values */
int fd;          /* file descriptor for Unix IO */
char *fname;     /* file name character string */
FILE *fp;    /* stream file pointer for buffered IO
             (FILE is typedef'ed in #STDIO.H file)*/
char rc;         /* returned character value */
```

9.2.1 Character type and conversion functions

b = isalnum(c)
: Returns true if input char **c** is alphabetic or numeric, i.e., 'a'-'z', 'A'-'Z', or '0'-'9'.

b = isalpha(c)
: Returns true if input char **c** is alphabetic, i.e., 'a'-'z' or 'A'-'Z'.

b = isascii(c)
: Returns true if input char **c** is an ascii value, i.e., 0 to 127 decimal.

b = isdigit(c)
: Returns true if **c** is numeric digit, i.e., '0'-'9'.

b = isxdigit(c)
: Returns true if **c** is hex digit, i.e., '0'-'9', 'a'-'f', or 'A'-'F'.

b = islower(c)
Returns true if **c** is lowercase alphabetic, i.e., 'a'-'z'.

b = isprint(c)
Returns true if **c** is a printable character (i.e., ' '-'~') and not a control character and not outside the ascii character sequence.

b = ispunct(c)
Returns true if **c** is an ascii character but is not alphanumeric and is not a control character.

b = isspace(c)
Returns true if char **c** is space character, i.e., blank ' ', tab '\t', newline '\n', carriage return '\r', or form feed '\f'.

b = isupper(c)
Returns true if char **c** is uppercase alphabetic, i.e., 'A'-'Z'.

rc = tolower(c)
Returns **c** converted to lowercase if possible, or returns **c** if no conversion possible.

rc = toupper(c)
Returns **c** converted to uppercase, if possible, otherwise returns **c** character unconverted.

9.2.2 String Manipulation functions

cp = strcat(cp1, cp2)
Returns the concatenation of string **cp1** with string **cp2**, in **cp1**. **cp1** must point to an area long enough for the result.

i = strcmp(cp1, cp2)
Compares two character strings. Returns a value less than 0 if **cp1** < **cp2**, returns a value greater than 0 if **cp1** > **cp2**, otherwise returns 0 to indicate

strings equal.

strcpy(cp1, cp2)
String **cp2** is copied into string **cp1**. **cp1** must point to an area large enough for the reuslt.

i = strlen(cp)
The length of the input string is returned. Length does not include the trailing '\0' character that defines the end-of-string.

9.2.3 Memory Management functions

The library offers a hierarchy of memory allocation and deallocation functions for several levels of efficiency and ease-of-programming considerations. Warning: be sure that allocated memory areas are returned to the free memory pool with the proper deallocation function. The **free** function is used for areas allocated with **malloc**, **alloc**, and **calloc**; the **retmem** function is used for areas allocated with **sbrk** or **getmem**.

For this discussion, the **#define NULL 0**, as found in STDIO.H, is assumed.

cp = alloc(i)
Allocates a memory area with **i** bytes, and returns a pointer to the allocated area. Return value is NULL if no area large enough could be found. The area is initialized to all zeroes. **alloc** calls **malloc**.

cp = calloc(i1, i2)
Allocates a memory area large enough to hold **i1** elements of **i2** size in bytes, i.e., allocates **i1** times **i2** bytes, and returns pointer to area. The area is initialized to all zeroes. The return value is zero (null pointer), if no space is available.

free(cp)
Returns or frees an area allocated by alloc, calloc, or malloc. Warning: do

not try to free areas allocated by **sbrk** or **getmem**, as no area length information is constructed for the **free** function by either. Use **retmem** instead.

cp = getmem(i)
getmem allocates a memory area of **i** bytes, and returns a pointer to the allocated area. Return value is NULL if no area large enough could be found. **getmem** searches a list of free memory areas first, then will call **sbrk** if no suitable area can be found on the free list. **getmem** does not record the size of the allocation, so areas can only be returned to the free list by calling **retmem**.

cp = malloc(i)
Allocates a memory area of **i** bytes, and returns a pointer to the allocated area. Return value is NULL if no area large enough could be found. **malloc** records the size of the area so that it can be returned and reused via the **free** function.

movmem(cp1, cp2, i)
Copies **i** characters from area pointed by **cp2** into area pointed by **cp1**.

retmem(cp, i)
Places an area allocated by sbrk or getmem on the free list. **cp** is the address of the area, and **i** is the size of the area in bytes. The area will be consolidated with adjacent free list entries if possible.

cp = sbrk(i)
Allocates **i** number of bytes of data memory, and returns pointer to allocated area. Return pointer equals -1 if no space was available. **sbrk** is the lowest level allocation function, and is used by **getmem**. **sbrk** does not examine the free area list, but instead 'pushes up' the high-water mark and allocates memory

not previously used or freed; thus, it should be used only as a last resort (no areas on free list). Areas allocated by **sbrk** can be returned to the free pool with **retmem**, but the high-water mark can not be decreased.

setmem(cp, i, c)
Copies character **c** into area indicated by **cp** for **i** number of bytes.

9.2.4 Standard IO Functions

Many of these functions will require that STDIO.H be **#included.** STDIO.H will provide **#define EOF -1** and **typedef FILE.**

exit(i)
Return to the *Instant-C* interpreter after closing any open files. The argument **i** is optional. If present, the value is passed to **_exit**, and control is returned to the interpreter. If the argument is not present, **_exit** is called with 0.

_exit(i)
Return to the interpreter directly, with no cleanup actions performed. [Currently, the argument **i** is not used.]

fd = open(fname, i)
Opens CP/M or MS-DOS file with name **fname**, and mode **i.** Mode may be 0 for reading, 1 for writing, 2 for read and write. The returned file descriptor is used for all subsequent accesses to this file until it is **closed.** The return value is less than 0 (i.e., EOF) if the file could not be opened, all file units are in use, or an input argument is incorrect.

i = close(fd)
Closes the file indicated by file descriptor **fd.** Return value is 0 if close is completed okay.

i = unlink(fname)
Deletes file **fname** from disk. Return value is zero if file is erased, otherwise -1 is returned.

fd = creat(fname, i)
A new file name with name **fname** is created on disk, and opened for output with file descriptor **fd**, which should be used for all subsequent operations on this file. If the file exists, it is deleted before creating and **open**ing the new file. The protection mode parameter, i, is not used. The return value is less than 0 if the file could not be created or opened.

i = lseek(fd, offset, origin) long offset; int origin;
fd is the file descriptor for an opened file. **origin** is a code that indicates the type of file positioning to be performed. **origin** values may be:

0	position after beginning of file.
1	position relative to current position.
2	position before end of file.

offset is the number of bytes by which the file position is changed. Be sure that a long value is passed to **lseek** for use as **offset**.

i = read(fd, buffer, i) char buffer[];
Reads **i** bytes from file **fd** into **buffer**. **buffer** should be large enough to accept the data. The value returned is the number of bytes actually read into **buffer**.

i = write(fd, buffer, i) char buffer[];
Writes **i** bytes from **buffer** to file **fd**. The return value is the number of bytes actually written.

fp = fopen(fname, cp)

Opens file with name **fname**, and returns pointer to a library allocated structure used for buffered file access via the various get and put functions listed below. **cp** is a pointer to a character string that describes the type of access to the file. The string "r" is for read access, "w" for write access, "a" for append access (writing at the end of the file). If the file cannot be opened, a null, or 0, pointer is returned.

Three files are implicitly open for *Instant-C* programs, and global file structure pointers are declared for these files: **stdin**, **stdout**, **stderr**. Currently, no 'shell' functions are provided to simulate the services of IO redirection found in Unix and Unix-like systems. Thus, **stdin** is input from the keyboard, and **stdout** and **stderr** are output to the console.

i = fclose(fp)

Closes file opened with **fopen**. For files opened for writing, any pending output is written to disk. **fclose** returns 0 if the writing of pending output and closing is completed successfully.

fd = fileno(fp)

Returns the the file descriptor given a file structure pointer, so that the Unix-compatible library functions may be used on files opened for buffered IO.

b = ferror(fp)

Returns true if an error has occurred while accessing or operating on a file.

b = feof(fp)

Returns true if the end-of-file has been reached on file **fp** by a read or write operation.

clrerr(fp) Clears the error and end-of-file flags for file **fp**. No get or put functions may occur on a file while the error flags are set.

i = putc(c, fp)
Puts or writes character **c** to file **fp**. Returns value less than 0 in case of an error.

i = putchar(c)
Writes character **c** to file stdout. Return value is less than 0 if an error or end-of-file has occured.

i = putch(c)
Write character **c** to console. Return value is character **c**.

i = fputs(cp, fp)
Writes a character string to file **fp**. The end-of-string '\0' is not written, and no new line ('\n') is supplied. Return value is less than 0 if an error or end-of-file has occured.

i = puts(cp)
Writes character string to file stdout. Unlike **fputs**, however, **puts** implicitly outputs a new line at the end of the string. Return value is less than 0 if an error or end-of-file has occured.

i = putch(c)
Writes character **c** to console. The character is returned.

cputs(cp) The string **cp** is displayed on the console. This function is implemented as a call to the operating system bdos(9, cp). The end of the string **cp** is first modified to be '$' instead of '\0', so the string should not itself contain '$'. The string end is set back to '\0' before return.

i = ungetc(c, fp)
"Backs up" file **fp** by one character. **c**

will be the next character read from file **fp** with any of the getc or gets type functions, below. **ungetc** will not work with **getch** or **cgets**.

i = getc(fp)
: Returns next character from input file **fp**. Return value is less than 0 if an error or end-of-file has occured.

i = getchar()
: Returns next character from file **stdin**. Return value is less than 0 if an error or end-of-file has occured.

i = getch()
: Returns character from keyboard.

cp = gets(s)
: Gets a line of input from file **stdin**. First character of string **cp** <u>must</u> be set to the maximum length of the string before calling **gets**. Return value is less than 0 if an error or end-of-file has occured.

cp = fgets(s, i, fp)
: Gets a line of input from file **fp**. Parameter **i** is the length of the character string **s**. A pointer to the string is returned if all went well, otherwise 0, or null pointer, is returned to indicate an error.

cp1 = cgets(cp)
: Gets a line of input from the keyboard. By calling bdos(10, cp), **cgets** takes advantage of any system-implemented input editing. **cp[0]** <u>must</u> be set to two less than the maximum length of the string **cp**. The input line will be returned, but will always be **cp+2**. The input will be terminated with a '\0' so that it can be treated as a normal C string. **cp[0]** on return will still be the space available for input, in characters, and **cp[1]** will be the number of characters in string **cp1**.

9.2.5 Formatted IO functions

Two types of formatted IO functions are provided: **printf** for output and **scanf** for input.

The **printf** functions format output according to a format control string. Several variations of **printf** may be used depending on where you want the output to go (see explanations of **printf**, **fprint**, and **sprintf**, below). The control string is output character-by-character until the end of the control string. The percent ('%') character causes special interpretation of the control string. Following the percent character is an optional field width and precision, specified as in K & R. The character following a % (or the optional field width and precision) describes how the next argument in the variable argument list is to be handled. The specifiers are:

c — the argument is treated like an ascii character.

d — the argument is treated like an integer and output in decimal.

e — a float or double argument is output in exponential form.

f — a float or double argument is output in fixed point decimal form.

g — a float or double argument is output in whichever of **e** or **f** format requires the least space.

o — the argument is treated like an unsigned integer, and output in octal.

s — the argument is treated like a pointer to a character string, and the string is output in ascii.

u — the argument is treated like an unsigned integer, and output in decimal.

x — the argument is treated like an unsigned integer, and output in hexadecimal.

l — the modifier **l** is placed before the specifier in the control string to indicate that the value has the size of a long integer. This is meaningful for the **c, d, o, u**, and **x** specifiers.

Any other character following a % is output as is, and does not alter the selection of the next argument for output. See K&R for more detail.

For the examples, **control_string** is a character string argument.

i = printf(control_string, ... args) Performs the general output formatting described above; output goes to **stdout.**

i = fprintf(fp, control_string, ... args ...)
Same as **printf**, but output goes to file **fp.**

cp = sprintf(cp, control_string, ... args ...)
Same as **printf,** but output goes to character string pointed by **cp.** The output string must be long enough for the output -- no checking by the library is possible.

The **scanf** functions are the inverse of **printf,** i.e., they interpret characters from an input stream and store the converted values via a list of pointers. Several variations are available (**scanf, fscanf**, and **sscanf**), depending on what input source you want to use. In the control string, blanks, tabs, and other 'white space' characters are not significant. Percent signs indicate a conversion specification, as detailed below. Any other characters indicate that a literal match must be made with the input stream, or the scanf is aborted. The input conversion specifiers consist of an optional field width (as a decimal integer), and a single character to indicate the type of conversion as follows:

c	copy single character from input, and store through next pointer into character. Unlike all other specifiers, no leading blanks are discarded from the input stream.
d	treat input characters as decimal number, and store integer value through next pointer in argument list.
e	treat input as floating point number. Notation may be fixed point or exponential.
f	treat input as floating point number. Notation may be fixed point or exponential.
o	treat input as octal number, and store unsigned integer value.
s	treat input as string, and store characters through next pointer in argument list, appending end-of-string '\0' character. Be sure that string is long enough for any possible input.
x	treat input as hex number, and store unsigned integer value.
*	followed by one of the specifiers above, asterisk means to perform the conversion, but don't store value.
l	followed by one of the specifiers above, indicates that a long value is to be stored (e.g., long value for **d** specifier, double value for **f**).

Remember that the argument list should consist of pointers. The return value is the number of stores that were completed before a mismatch of specified format and input occurred, or before the control string was exhausted, or before the argument list was exhausted. Should end-of-file be reached in the input stream, a negative value is returned. See K&R for more details and examples. Currently, leading 0x hex numbers are not implemented.

```
i = scanf(control_string, ...  args ...)
```
scanf performs the general input formatting functions described above, using the stream **stdin** for input.

```
i = fscanf(fp, control_string, ...  args ...)
```
Same as scanf, except that input is read from file **fp**.

```
i = sscanf(cp, control_string, ...  args ...)
```
Same as scanf, except that input is read from string **cp**.

9.2.6 Low Level Routines

```
c = bdos(function_number, arg)
```
Invoke the operating system directly. The **function_number** argument is passed in CL for CP/M-86, and AH for MS-DOS and PC-DOS. In both cases, the second argument is passed to the operating system in register DX. The value of this function is the value of register AL after the operating system returns.

```
i = bdosw(function_number, arg)
```
Same as bdos, except that the value returned is a 16-bit register value. For MS-DOS or PC-DOS, this is the AX register. For CP/M-86, this is the BX register.

The following built-in functions provide direct access to your system's hardware resources, and remove much of the need for any assembly language programming with *Instant-C*. Errors in the use of these functions can be catastrophic; you should use these functions only if you have a thorough understanding of the hardware operations involved.

Assume the following declarations for the discussion of low-level functions:

```
int port_number;     /* hardware IO address */
int i;               /* miscellaneous value */
unsigned seg, offset;  /* segment:offset
                          for full address */
```

```
unsigned cpuflags;   /* processor flags
                        register value */
REGS inregs, outregs;  /* structure for
                        cpu data registers */
int intno;           /* system interrupt number */
SREGS segregs.       /* structure for cpu
                        segment registers */
```

The REGS and SREGS typedefs are found in INTLIB.H.

c = inportb(port_number)
: Do an input byte instruction to the specified port for your 8088/8086 processor. This function allows you to read from any input device in your computer.

outportb(port_number, i)
: Do an output byte instruction to the specifed port for your 8088/8086 processor. The byte **i** is send to the output device. Together with **inportb**, this function makes it possible to do the very lowest-level I/O in *Instant-C*.

cpuflags = _call(seg, offset, &inregs, &outregs)
: Perform a long call to location **seg:offset**. CPU registers are first loaded from the structure **inregs** before the call, and saved in structure **outregs** after the call. The processor flags are returned as the value of **_call**.

cpuflags = _flags(new_cpuflags)
: The processor's flags register is set to **new_cpuflags**. The prior value of the processor flags register is returned as the value of **_flags**. The arithmetic flags (overflow, carry, etc.) are unlikely to be meaningful. The single-step instruction trap flag cannot be set with **_flags**. This function does have use, however, in controlling whether external interrupts are allowed or disabled.

cpuflags = _interrupt(intno, &inregs, &outregs)
: The hardware interrupt number **intno** is

signalled after loading the CPU registers from the structure **inregs.** Upon return from the interrupt, the CPU registers are saved in the structure **outregs** and the processor flags register is returned as the value of **_interrupt.**

_movdat(sseg, soffset, dseg, doffset, i)
Data is moved from **sseg:soffset** to location **dseg:doffset,** for **i** bytes. Thus any location in your system's memory may be accessed. **_movdat** moves data by ascending addresses, and does not check for any overlap of the source and destination data areas.

_segread(&segregs) SREGS segregs;
The segment registers (CS, ES, DS, SS) are copied into the structure **segregs.**

_() You read it right, it's _(). This function is an explicit breakpoint interruption. It can be placed at any point in your program to force control to the *Instant-C* interpreter for debugging. For example:

```
if (var < 0 || var > 9)
    _();    /* breakpoint when
out of range */
```

9.2.7 Interrupt Support Functions

These functions are supplied in the file INTLIB.IC, and are not built into the IC program. To access them, you will need to #include intlib.ic. [Note: these functions and there specifications are still preliminary and subject to change.] For discussion purposes, the following declarations are assumed:

```
int intno;  /* an interrupt number, 0-255 on 8086 */
struct _int_vector vector;  /* image of an interrupt
                               vector, i.e., IP:CS */
unsigned flags;      /* processor flags register
                        value */
int (*handler)();    /* a user-written interrupt
                        handler function variable */
```

```
struct _int_prologue *ip;  /* pointer to an
                              interrupt handler prologue */
```

interrupt_get(intno, vector)

Copy the interrupt vector (IP and CS) for interrupt number **intno** to structure **vector.** This is useful in saving interrupt vectors for later restoration with **interrupt_set**.

interrupt_set(intno, vector)

Copies the structure **vector** to the system interrupt vector number **intno.** This can be used to restore system interrupts that you overwrote with **interrupt_install**, or can be used to switch between different interrupt handlers that share the same interrupt number.

ip = prologue_init(handler, flags)

An interrupt prologue must be constructed to call your *Instant-C* interrupt handler. The prologue performs such services as setting the segment registers to address your data, switching to the *Instant-C* execution stack, and setting the processor flags register to the value of **flags.** Several prologues may call the same **handler** function.

interrupt_install(intno, ip)

Installs your interrupt handler function prologue **ip** in system interrupt vector **intno.** After calling **interrupt_install,** the occurance of an interrupt number **intno** will result in a call to your **handler** function. The handler can be switched or de-installed with the **interrupt_set** function.

Interrupts may occur, and the handlers execute, while you are modifying or executing (other) *Instant-C* functions. Handlers for clock or keyboard or other hardware-generated interrupts are examples of interrupts that may occur at any time as long as the handler is installed for the interrupt. Only basic memory allocation (**sbrk**) is protected from interruption, so it is okay to allocate memory with **sbrk** in an interrupt handler, but it is not okay to **alloc** or **free** within a handler if there is any chance that it will be handling an interrupt which occurred during another function's call to **alloc** or **free**. Other things to avoid: don't update an attached handler function with the editor, and don't **quit** from *Instant-C* without reinstalling the system default handler. DOS functions may not be re-entrant, so asking for terminal input in a handler called by an interrupt during terminal input wait is likely to fail, as is calling bdos for terminal output from a break interrupt.

Save your code to disk before testing interrupts or other hardware functions: system hangs are likely to occur.

In general, you should revert an interrupt vector to its state as found before installing an *Instant-C* coded handler. The functions interrupt_get and interrupt_set make this easy. Interrupts, particularly if signalled by hardware or some agency independent of *Instant-C*, should be reverted before doing anything that could interfere with your handler's execution, such as issuing the **quit** command. All *Instant-C* interrupt handlers should return, and should not call **exit**, **_exit**, or longjmp. Further, handler functions should not fault (*e.g.*, stack overflow, divide by zero) unless an appropriate handler has been installed for that interrupt also. Breakpoints, single-steps, and control-Break handling will not occur for handler functions active due to interrupt. Handlers execute on the *Instant-C* execution stack, and thus have full addressing capability, and may call any other function.

Appendix A

How Instant-C Differs from Standard-C

This appendix contains descriptions of the extensions available, together with an enumerated list of all of the standard features not yet implemented.

A.1 Extensions

1. Version 7 **void** type for functions which don't return a value.
2. Integrated source-language debugger including single-step by statement tracing.
3. Lint-like error checking for the number of arguments and size of argument lists.
4. Automatic formatting of all functions and declarations.
5. Immediate execution mode (type in a C statement and it executes).
6. Integrated full-screen editor for rapid syntax error correction.

A.2 Features Not Yet Implemented

1. Initializers for struct's and unions.
2. Initializers for auto and register variables.
3. Initializers for pointer variables.
4. Packed fields in structures (bit-fields).
5. String literals which extend past end-of-line
6. Enums
7. #define's more complicated than constant expression
8. #define's with arguments
9. #undef
10. Assembly language interface
11. Math functions in library (trig, log, etc.)
12. #line
13. Obsolete assignment operators (=+, =-, etc.)

Appendix B

Error Messages and Explanations

Instant-C has a rich and extensive set of error messages designed to help you understand exactly what *Instant-C* thinks is wrong. This appendix includes all of those error messages together with additional information about each error. The errors are sorted alphabetically by the first word in the message.

Error messages are displayed either at the top of the Instant-ED screen, above the status line, or prefixed by "**ERROR: ". Any error message which starts with two asterisks, (and is not followed by ERROR:), is an internal consistency check message, and indicates a problem with *Instant-C* rather than your program.

B.1 Language Errors

The following error messages indicate that you have made a mistake in your program or have used a feature that isn't implemented yet.

<name> **cannot start a statement**

Instant-C was expecting a statement, but found the **<name>** instead.

<name> **has no procedure code.**

You have typed a command which needs a defined function as its argument. *Instant-C* doesn't recognize the name given. Misspellings are the most likely cause of this error.

"<name>" invalid in function header

Instant-C was processing the formal parameter list in a function header and found something other than a comma, a name, or a right parenthesis.

<name> is already in the dictionary

You have tried to rename a variable or function to a name which is already defined as something else.

<name> is not a function

You have tried to call a function, and the **<name>** is not a function. This error may be caused by a missing operator in a parenthesized expression.

<name> is not a member of struct/union <name2>

The name following a . or -> selector operator is not a member of the indicated structure or union. This error can be caused by spelling errors -- either at the point of this error or in the template for the struct or union.

<name> is not a parameter of this function

Instant-C does not recognize the parameter **<name>** in the argument declarations for this function. This error is usually caused by a misspelling, either in the function header or in the argument declaration. It can also be caused by a missing left brace at the start of the function.

<name> is not in the dictionary

You have typed an *Instant-C* command such as **#rename** and the name argument to the command doesn't exist. Misspellings can often produce this error.

Addition of two pointers

You have tried to add two pointers together, an operation which is not defined in the C language.

Arithmetic on function or array pointer

You are trying to do pointer arithmetic on a pointer to an array or function. Since *Instant-C* doesn't know how big the pointed to array or function is, it can't do the arithmetic.

Arrays of functions are not supported

You have tried to declare an array of functions, which is not supported in **C**. You may want to declare an array of pointers to functions, which is supported.

Attempt to subscript non-array and non-ptr

You have tried to apply the subscript operator ([) to an expression which is neither an array name, nor a pointer.

BREAK not valid outside of loop or switch

Instant-C found a **break** statement which appears to be outside of any loop or switch statement. Check for misplaced braces.

Buffer empty; no changes made

You have used **Ctrl-F** or the **F** command to leave Edit Mode, but there was no text in the buffer to compile. *Instant-C* switches to Command Mode, and does not save any compiled functions or data.

Call on non-function expression

You have tried to call a computed function address, and the computed expression does not address a function. This error may be caused by a missing operator in a parenthesized expression.

Called undefined function <name>; aborted

You have invoked a function which, directly or indirectly, called another function which has not yet been defined. This can be caused by misspellings, or because you forgot to include a source file.

Can't apply . if not struct/union

Can't apply -> if not struct/union

You have used one of the struct/union member selection operators on an expression that is neither a structure nor a union.

Can't apply -> to non-pointer

The expression to the left of the -> operator should be a pointer to a struct or union and isn't.

Can't create <name> for module

You have issued a **#savemod** command, but *Instant-C* couldn't open the file for writing. This error might be caused by full disks or disks that are read-only.

Can't create file <name>

Instant-C can't open the outfile file you requested in a **#save** command. Possible causes are a full disk or a read-only disk.

Can't increment/decrement by pointer

You have tried to use a pointer expression as the right operand of the

+= or -= operators.

Can't open output file "<name>"

Instant-C was unable to open a file for output, possibly because the disk was full, or because the disk was read-only.

Can't parse <name>

Instant-C doesn't know what to do with the first word on a line in command mode.

Can't store into expression

Instant-C thinks that the left operand of an assignment operator is an expression that doesn't have a valid **lvalue.** This can be caused by omitting a unary * indirection operator.

Can't use . if not simple struct or union

The . member selection operator can only be applied to struct or union lvalues. Your code uses a more complicated expression. Possibly caused by a missing * (indirection) operator.

Constant expression overflow processing <name>

The constant expression indicated has overflowed *Instant-C*'s internal tables. You can get around this limitation by breaking up the expression using multiple **#define**'s. We believe that this error should never occur in normal programs, so please notify us if you get this error -- we would like to see your program.

CONTINUE not valid outside of loop

Instant-C found a **continue** statement which appears to be outside of any loop or switch statement. This error can be caused by misplaced braces.

Editing aborted; changes not made.

You were editing program or function text, and gave the **Ctrl-Q** or Quit commands. *Instant-C* has thrown all of your text away, and has not made any changes to your programs.

else not following if () statement

Instant-C found an **else** which isn't connected to a previous if statement.

Error writing <name>; aborted (disk full?)

Instant-C encountered disk error while writing a module file. This error is usually due to a disk filling up.

Error writing header for <name>; aborted

Instant-C encountered disk errors writing the module header while it was trying to execute a **#savemod** command. This error is usually due to a full disk.

Function returning array not supported

You have tried to declare a function which returns an array as its value. You should check the parenthesis structure in your declaration. You may also want to return a pointer to an array, which is the only way array values can be returned in C.

Function returning function not supported

You have tried to declare a function which returns a function as its value. You should check the parenthesis structure in your declaration. You may also want to return a pointer to a function, which is the only way function-related values can be returned in C.

getmem: no storage

Instant-C has run out of storage for its internal tables. Save your programs to disk and start a new *Instant-C* session by quiting, and giving another IC command.

I'm sorry, but I don't know the word "<name>"

You have used a word that *Instant-C* doesn't recognize. This is most likely a misspelling either at the point of this error or in the previous declaration for this word.

Ignoring unfinished #define definition

You have entered the pre-processor directive **#define** as the last word in your input file. Fix this by removing the **#define** or by adding a word to be defined.

Ignoring unfinished data declaration

Instant-C was expecting a name in a data variable declaration, and found something else instead.

Ignoring unfinished Object definition

Instant-C encountered an end-of-file while processing a **defobj** command.

Incomplete expression

Instant-C encountered an end-of-file while parsing an expression.

Indirection not on pointer

The right operand of a unary * (indirection) operator is not a pointer.

Insufficient code buffer space left

Instant-C doesn't have enough space left to create the buffers it needs to do code generation. Save your source programs, and start a new *Instant-C* session.

Local typing is too complex

You have entered a complicated declaration that has more levels of attributes than *Instant-C* is able to handle. This error is very unlikely; therefore, if you get it, check your source for other errors in the indicated declaration.

Missing (in function definition header

Instant-C was expecting a left parenthesis as part of the definition of a previously referenced function.

Missing (in function header

Instant-C was expecting a left parenthesis in a **void** function definition header.

Missing close quote in char literal

You have typed a character literal, but the trailing quote is missing.

Missing comma in function call

You have omitted the comma between arguments in a function call.

Missing formal argument name

Instant-C was expecting the name of another formal parameter in the header of a function definition. This error can be caused by omitting the right parenthesis in the header.

Missing inital quote in char literal

Instant-C was expecting a character literal, and didn't find the initial **'**.

Missing initial quotes in string literal

Instant-C was expecting a string literal, and didn't find the initial **"**.

Missing left parenthesis

Instant-C was expecting a left parenthesis after an **if** or **while**, but didn't find it.

Missing left parenthesis in function call

Instant-C was expecting the left parenthesis starting the argument list in a function call.

Missing member name after .

Missing member name after ->

Instant-C was expecting a member name after the struct/union selector operator.

Missing name in declaration

Instant-C didn't find the name of the object you were declaring in this declaration.

Missing parenthesis

Instant-C can't find the left parenthesis that is supposed to follow a **for**.

Missing quotes at string end

Instant-C reached the end of a line while processing a string literal. *Instant-C* does not allow string literals to be continued over more than one line.

Missing right bracket

Instant-C expected a right bracket]. This error can occur in array declarations. It may be caused by an invalid expression for the array size.

Missing right bracket (])

You have subscripted a pointer or pointer expression, and *Instant-C* can't find the right bracket (]) where it expects to. This error may be caused by a syntax error in a subscript expression.

Missing right bracket for <name>

Instant-C expected a right bracket to finish the subscripting of **<name>**. This error is usually caused by a syntax error in the expression for the subscript value.

Missing right parenthesis

Instant-C expected a right parenthesis). This error can occur in complex declarations, function declarations, **for** statements and in other constructs.

Missing right parenthesis in function call

Instant-C expected the right parenthesis which terminates the argument list for a function call.

Missing right parenthesis in function declaration

Instant-C was processing a function declaration and was expecting a right parenthesis immediately after the left parenthesis. This error can be caused by having too few right braces in the preceding function definition.

Missing semicolon

Instant-C is expecting the semicolon between the control expressions in a **for** statement header. This may be caused by an invalid expression.

Missing semicolon (;) before "<name>"

Instant-c thinks that it has completed parsing a statement, and expects to find the semicolon to terminate the statement.

Missing while in do-while statement

Instant-C can't find the **while** keyword at the end of a **do** loop. This error can be caused by misplaced or missing braces.

Name <name> is not a constant

You have used a name that is not a **#define**'d constant in a place where *Instant-C* was expecting a constant expression. This error may be caused by *Instant-C*'s limitation that **#define** expressions must be constants.

Neither tag nor template for struct

Neither tag nor template for union

Instant-C expected a tag name or a left brace (starting a template) after the word **struct** or **union**.

No file name given

No file name was given in a **#save** command, and there is no default name from a previous **#save** command or from the initial *Instant-C* command line.

No more function space left; function not updated

Instant-C has run out of space to update or create new functions. The editing you just did cannot be saved. Save your source programs on disk, and start a new *Instant-C* session.

No previous object to edit

You have given an **ed** command without a name to edit. Unfortunately, there is no name to use from previous edits (because this is the first **ed** command of your session).

No space to create function

Instant-C has run out of space to create new functions. Save your source programs, and start a new session.

Not call on pointer to function

You have entered a computed function address call, but the computed expression does not result in a pointer to a function. This error may be caused by a missing operator in a parenthesized expression.

Object is not currently open

You have entered a **endobj** command without previously issuing a **defobj** command for the same object.

Old file <name> already exists; aborted

You have given a **#save** command naming or implying a file that already exists on the disk. *Instant-C* will ignore the command.

Out of input, check unterminated remark

Instant-C encountered the end of input while trying to parse a statement. This error may be caused by an incorrectly terminated remark or string literal which has swallowed a right brace (}).

OUTFILE: invalid destination

Instant-C didn't understand the name you gave as the destination of an **outfile** command. The valid names are: **printer**, **crt**, or **both**.

Pointer cannot be left operand of <operator>

You have tried to use a pointer expression as the right operand of an assignment operator.

Pointer cannot be right operand of <operator>

You have tried to use a pointer value as the left operand of an assignment operator other than =, +=, or -=.

Premature eof in constant expression

Instant-C was processing a constant expression (in a **#define** or array dimension) and reached the end of input.

Premature EOF while parsing arg declarations

Instant-C was parsing the argument declarations of a function when it ran out of input.

Ran out of input while compiling (Missing } ?)

The code compiler has reached the end of your program or source file, but does not have a complete function or data definition. This error can be caused by a missing right brace in some cases.

Redefining data <name> as function

You have entered a new function definition, but there is already a data variable with the same name.

Remark space overflow

Instant-C has run out of space to store remarks from your programs. Save your source programs, and restart a new session.

Rename: missing name

You have omitted the name to be changed in a **#rename** command.

return only valid in function definition

The **return** statement may not be typed at command level, as there is no function active to return from.

sizeof constants not implemented yet

Instant-C does not yet handle **sizeof** in constant expressions.

Sorry, but "<name>" has no source to edit

You have tried to edit something other than a function or data variable, and *Instant-C* doesn't know how to create a source version of the object. This message would result from typing "ed ,".

Sorry, but <name> cannot start a declaration

Instant-C was expecting a declaration, but you have entered a name, statement, or command instead. This error can be caused by #including a file with executable statments, or by leaving out the initial left brace ({) in a function definition.

Sorry, but <name> is not implemented yet

You have used a valid C-language syntax that is not yet handled by *Instant-C*.

Sorry, can't find file **<name>**

You have named a file in an **#include** or **#infile** command which is not in the currently attached disks. The command is aborted.

Struct/union **<name> already has template**

You have defined the structure's and/or union's template (the declarations enclosed in { }) more than once.

Struct/union **<name> has no template yet**

You are trying to select a member of the named aggregate with the . or -> operator, but no list of members has been defined in a template for the struct or union yet.

Subtraction **of dissimilar pointers**

You have tried to subtract two pointers which do not point to the same type of object. *Instant-C* does not know how to scale the resulting difference.

Subtraction **of pointer from integer**

While *Instant-C* allows you to subtract an integer from a pointer, subtracting a pointer from an integer is not defined in the C language.

Symbol <name> **is already a member of struct/union**

You have used the same member name for two different elements of a struct/union.

Symbol <name> **is already a tag (struct/union)**

You are trying to define a new union which is already defined as a struct (or a new struct which is already a union).

The name <name> is already declared as something else

You have declared the same name more than once. Since this error occurs outside of a function definition, it can sometimes be caused by incorrect nesting of braces in functions.

Too few args in call to <name>

You have tried to call a function with fewer arguments than it was compiled to receive.

Too many args in call to <name>

You have tried to call a function with more arguments than it was compiled to receive.

Unterminated remark swallowed program

The closing */ is missing from a remark. As a result, *Instant-C* has read all the way to the end of your program. The cursor should point to the beginning of the remark.

B.2 Internal Errors

The following messages all start with two asterisks and indicate an internal error or bug in *Instant-C*. If you receive one of these error messages, please report it to Rational Systems, Inc. so that we can fix the problem. Any copy of the program causing the error or the sequence of commands that resulted in the problem will help us track down the bug.

****<name>: insufficient space for buffer**

****(<name> is not a property)**

****ADDCODE: procedure table full**

```
**addfunct: broken chain

**addliteral: broken chain

**addprmpt: too many prompts

**Call relocation buffer overflow!!

**Can't generate code for <name> <name>

**Char literal unfinished at line end

**Code buffer overflow!!

**codefuse: overlapping storage

**delsym: symbol <name> not found

**dropprmpt: unmatched prompt

**Duplicate procedure table entry

**Early escaped eos

**GENREAD: no code to generate

**internal error ******

**Internal error in procargs

**Instant-C restarted

**Invalid expression -- not declared

**Invalid size for increment or decrement

**level error <#>, object is <name>.

**litvalue of non-literal

**makecode: overlapping code

**memfuse: overlapping storage

**Name <name> is missing during initialization

**No code for subscripting
```

**No code to generate for <operator>

**No DATAADDR property -- invalid code generated

**No generation proc for <name>

**No leftgen for <operator>

**No object code for <name>

**No rightgen for <operator>

**No size info for <type_name>

**No stack object found

**No type info to process <operator>

**null h_flast component

** null inproc **

**Null object to addprop <name>

**Object <name> doesn't have address code

**Object <name> doesn't have value

**Premature termination of file list elements

**PROPVAL with null clist

** recursive code generation

**Recursive remark handling

**Relocation buffer overflow!!

**SETPROP with null clist

**String code generation buffer overflow!!

**There is no object code for <operator>

**UGEN: no code to generate

Appendix C

Summary of *Instant-C* Commands

This appendix contains a list of all of the commands available to the *Instant-C* interpreter, together with a brief description.

In addition to the listed commands, any C language expression or statement can be issued as a command. If the expression has a value, i.e. is not a call on a **void** function, its value will be printed after evaluation.

All commands are available with and without the leading '#' in the name: both forms are provided in case you have a routine with the same name, e.g., **rename.**

C.1 User Commands

#back
: Display a trace back, showing all functions called to the point of an interruption in execution.

#delete *name*
: Deletes *name* from the current memory file.

#dir *d:filename.ext*
: Display directory. *filename.ext* may include * and ?. #index(#dir command)

#ed
: Edit the last function, data, or **#define**'d literal edited.

#ed *name*	Edit the function or data or **#define**'d literal. If *name* doesn't exist yet, it is assumed to be an **int** function.
#erase *d:filename.ext*	Erase file from disk.
#go	Resume execution of an interrupted program.
#infile "filename"	Read and execute interpreter commands from the named file.
#list *name*	Display the C-language source for the named function, data variable, or #defined literal.
#list	Display the C-language source for the entire contents of the current file.
#listfile	Display the memory files loaded into *Instant-C*.
#listname	Display the names of data variables, #defines, functions, and #included files found in the current memory file.
#llist *name*	Print the C-language source for the named function, data variable, or #defined literal on the standard printer.
#llist	Print the C-language source for the entire contents of the current file and the standard printer.
#load *d:filename.ext*	Read and compile the file into your *Instant-C* workspace.
#local *function_name*	Interpreter command line expression evaluation occurs in the context of *function_name*. If *function_name* is omitted, will display the current function context, if any.

#make *filename start_function*
Create a disk file to be astand-alone version of your program. If *start_function* is omitted, execution of the program will start with the **_main** function.

#new *memory_file*
Creates a new memory file, or clears the current memory file if no *memory_file* is specified.

#outfile {printer | crt | both}
Redirect the output from *Instant-C* to the indicated device. **outfile printer** is useful for making transcripts of *Instant-C* sessions.

#pc *location count*
Display memory, in character format, for *count* characters.

#pd *location count*
Display memory, in decimal format, for *count* words.

#po *location count*
Display memory, in octal format, for *count* words.

#ps *location count*
Display memory as character string pointers, for *count* words.

#pu *location count*
Display memory as unsigned decimal integers, for *count* words.

#px *location count*
Display memory, in hex format, for *count* words.

#quit
Terminate the *Instant-C* session and return to your operating system.

#rename *oldname newname*
Change the name of an object in your C program.

#reset *level_number*
Dispose of interrupted execution environments, back to the optional *level_number*.

#run *command_arguments*
Execute your C program as though it were invoked from operating system command level. Execution will start with **main(argc, argv)**.

#save *filename*
Write the current memory file to a disk file. The disk file must not exist yet.

#savemod *filename*
Create a new copy of *Instant-C*, together with the current workspace. This command can be used to customize *Instant-C*, or to save your work in progress.

#segments
Display the address, maximum size, and current used space in each of the data areas managed by *Instant-C*.

#shell
Execute one or more operating system commands under PC-DOS or MS-DOS.

#step
Resume execution for one statement.

#step exec *C statement*
Executes the *C statement/expression* and stop after the first statement is executed.

#step in
Resume execution for one statment, without stopping in any nested functions that may be called.

#step out
Resume execution until the next function is called.

#step return
Resume execution until the current function returns.

#system — Terminate the *Instant-C* session and return to your operating system.

#trace *function_name*
Breakpoints will occur for the function upon call and return.

#type *d:filename.ext*
Display disk file on screen.

#untrace *function_name*
No more function call breakpoints will be issued for the function.

#use — Display the name of the currently active file (the one that will be written if a **#save** command is given with no name

#use *name* — Switch to a different file in the workspace.

C.2 Internal Commands

The following commands are currently available to help debug *Instant-C*. They may not be in the product when it is released for the general public.

#dsym function_name
Display the arguments and local variable of the named function, struct, or union.

#dproc function_name
Display the compiled code for the named function.

#edconfigure
Read in the configuration files to reconfigure the built-in *Instant-C* editor. This is similar to doing an **ed @** command for the stand-alone editor.

#fsource name
: Display the internal symbol table for the named object.

#resetint
: Prepare system interrupt vectors for interrupts used by *Instant-C* internally. Use after setting variable **_intnumber**.

#time
: Display the current time and date.

#variable name
: Allow a variable number of arguments when calling function **name.** This command is provided to support printf and scanf in the library.

#what hex_address
: Display the name of the object whose symbol table entry is at the indicated address.

#where name
: Display the hex address for the symbol table entry for the indicated name.

Appendix D

Language Summary

This appendix contains a brief list of all the features in Instant-C, presented in the same order as chapter 13 of K&R.

(This appendix will be provided in a later release.)

Appendix E

How to Install *Instant-C*

This appendix contains any special changes or instructions necessary to install *Instant-C* or to make customizations to *Instant-C*.

Before starting to use *Instant-C*, make at least two working copies of the distribution diskettes, and put the original copies in a safe place. If you are working in a single floppy disk environment, you will need to install your operating system on the disk so as to make your *Instant-C* working disk "bootable".

If you are NOT using *Instant-C* on an IBM-PC or compatible, you should refer to the sections below about "Configuring the Keyboard" and "Configuring Screen Output". These sections describe the steps necessary to adapt *Instant-C* to your computer and terminal.

E.1 List of Distributed Files

The following files should be present on your *Instant-C* master disks:

MS-DOS or PC-DOS Version

IC.EXE The *Instant-C* program with libraries. When you are running *Instant-C*, this is the only file you will need.

ED.EXE The stand-alone version of the *Instant-C* editor.

ICBASE.EXE The base *Instant-C* program without libraries. You need this to create an *Instant-C* for small memory space or with completely different libraries.

SCREEN.EXE The screen configuration program. If you have an IBM PC or PC/XT or compatible, you don't need this.

KEYBOARD.EXE The keyboard configuration program. You can use this program to change the keyboard usage of *Instant-C*'s editor.

CP/M-86 Version

IC.CMD The *Instant-C* program with libraries. When you are running *Instant-C*, this is the only file you will need.

ED.CMD The stand-alone version of the *Instant-C* editor.

ICBASE.CMD The base *Instant-C* program without libraries. You need this to create an *Instant-C* for small memory space or with completely different libraries.

SCREEN.CMD The screen configuration program. If you have an IBM PC or PC/XT or compatible, you don't need this.

KEYBOARD.CMD The keyboard configuration program. You will use this program to change the keyboard usage of *Instant-C*'s editor.

All Versions

README.DOC	Special or late documentation not found in this manual.
HELLO.C	The simple test file to demonstrate that *Instant-C* is working.
LIB.IC	The master source file for the libraries in *Instant-C*. It includes the other **.IC** files.
CTYPE.IC	Library source file containing the character classification and transformation functions.
STRLIB.IC	Library source file containing the string handling functions.
MEMORY.IC	Library source file containing the memory allocation functions.
STDIO.IC	Library source file containing the file and system input/output functions.
PRINTF.IC	Library source file containing **printf**, **scanf**, and their supporting routines.
FUNCVAL.IC	Library source file containing expression display functions.
INTLIB.IC	Library source file containing interrupt handling and signaling functions.
LS1.C	The library source in **CTYPE.IC**, **STRLIB.IC**, **MEMORY.IC**, **STDIO.IC**, **PRINTF.IC**, and **FUNCVAL.IC**. This file is included to simplify the building of a standard library version of *Instant-C*.
STDIO.H	This is the header file that declares commonly used objects, such as the FILE typedef for stream file IO, the #define

for NULL, etc..

INTLIB.H This is the header file for use with INTLIB.IC. It contains structure definitions such as REGS and SREGS (used with the **_call**, **_interrupt**, and **_segread** functions).

ICEDSCRN.CFG
The screen drawing configuration file. It is created by the SCREEN program and read by the editor during configuration. As delivered, this is a configuration for an ANSI standard/VT-100 terminal. This file is not used for IBM PC versions.

ICEDKEYB.CFG
The keyboard configuration file. This file is created by the KEYBOARD program and read by the editor during configuration. As delivered, this is a config uration for an IBM PC keyboard, and uses the PC function keys.

ICEDKEYB.MNU
This is a data file used by the KEYBOARD program to name all of the functions possible in the editor.

E.2 Configuring Screen Output

If you are using *Instant-C* on an IBM PC or PC/XT or compatible, you don't need to configure the screen output and should skip this section.

Since *Instant-C* uses full screen operations, it is necessary to tell *Instant-C* how to draw and perform various functions on the screen. You can do this by running the program we have provided, SCREEN. SCREEN is an interactive application which builds or modifies a configuration file containing screen-driving character sequences. The configuration file (ICEDSCRN.CFG) is then read by

Instant-C to customize the screen editor to your terminal.

Note: a configuration file is provided for IBM PC or compatible machines. If you have one of these machines, you will not need to specify any screen operations. SCREEN will let you override the default screen attribute selections, however, should you want to do so.

To run the screen configuration program, simply type:

A>*screen*

The program prompts with a menu of choices and screen function. The normal way to run SCREEN is to type the number of the screen drawing functions, and answer the prompts. You will need the manual for your terminal to enter the proper sequence of characters. The 'T' tests only test what has been entered, so your strategy should be to get cursor addressing right, then clear screen and (if your terminal has it) clear to end of line. Then use 'T' after each change to verify your progress.

The 'B' for numeric base is for entering characters by their value, rather than the actual keystroke. (Some manuals use decimal, some octal, and some hexadecimal.) If any of your sequences use a carriage return, you will need to enter a numeric value for the CR, since SCREEN uses return as the delimiter to indicate the end of a sequence. BE SURE to use the 'T' command to test your configuration before writing it to disk. You can get some spectacular, but undesirable, effects if your screen configuration is wrong.

After running SCREEN to create your configuration file, you can configure the keyboard (see following section). After preparing the configuration files, you will need to build or modify *Instant-C* and ED for the changes to take effect.

E.3 Configuring the Keyboard

You can reconfigure the key function interpretation of *Instant-C*'s editor. This key reconfiguration is designed to adapt *Instant-C* to different computer keyboards and terminals. You can also use the key reconfiguration feature to make the *Instant-C* editor more like some other editor with which you are more comfortable. You can reconfigure the keys on an IBM PC or PC/XT computer if you so desire. See Appendix F for details about all of the key functions.

Note: you do <u>not</u> need to reconfigure the keyboard at all to use the *Instant-C* editor. This feature is provided solely to make the keyboard interface as useful as possible for you.

To reconfigure the keyboard, you should run the KEYBOARD program provided on your *Instant-C* master disk. The KEYBOARD program interacts with you and creates or modifies a file containing the keyboard assignments. The keyboard assignment file is then read by *Instant-C* to control which keys are bound to which editor functions.

If you are not using an IBM PC or compatible, you must run SCREEN, the screen configuration program, before running the keyboard configuration.

To run the keyboard configuration program, simply type:

A>*keyboard*

The program prompts with a menu of choices. You can type 'H' and get the menu back again. The most frequently used command is 'B' for defining key to function bindings. Key bindings can be overriden just by redefining them. You can map several keys to the same function if you wish. Use the 'D' (for delete) command to eliminate a key binding completely.

With Bind, you essentially *teach* the program what keys or sequence of keys are to perform what editor function. When defining, the keys or sequences of keys can be entered just as you would type them when running the editor, followed by a return. Note: some ASCII terminals have function keys which send several characters at a time and include a carriage return. If this is true for your terminal, use the 'S' option to set the delimiter to some other character ('/' is a good choice).

After running KEYBOARD to create your configuration file, you will need to build or modify *Instant-C* and ED. The new configuration takes effect when:

- You run an unconfigured editor (as in ICBASE). In this case, configuration files must be present in the directory for the editor to work.

- For *Instant-C*, you issue a **#edconfigure** command. See sections on building and modifying *Instant-C*, below, for instructions on how to make the configuration a permanent part of your *Instant-C*.

- For ED, you go through the editor configuration process ("*ed @*"). See "Configuring a New ED", below.

E.4 Building a New *Instant-C*

Instant-C is delivered to you in both of two forms:

1. A pre-configured form with the standard library built into the workspace and with the editor configured (IC).

2. A "raw" form, with no configurations performed (ICBASE). All components needed to reproduce the pre-configured are also provided.

Major changes to the standard library (particularly replacing,reducing, or eliminating it) are best accomplished by rebuilding *Instant-C* from scratch. Minor changes, such as changed editor configurations, modified or added library functions, or changes to default settings of options are best handled by "cloning" an *Instant-C* (described in "Modifying Your *Instant-C*", below).

To build a new *Instant-C* from scratch, you need to have the following files:

ICBASE — .EXE for PC-DOS (and MS-DOS) or .CMD for CP/M-86. The raw *Instant-C* programs.

LS1.C — The source for the library files. (Processed to remove all comments to save data space -- future releases will use the *.IC files instead.)

STDIO.H — This is the header file with declarations used by the standard function library.

ICEDKEYB.CFG — The keyboard key assignment configuration file.

ICEDSCRN.CFG — The screen drawing configuration file.

Once you have collected (or created) all of these files on a single disk drive, you can build a new *Instant-C* by following these steps:

A>*icbase*

Start up the *Instant-C* base code.

\# *#edconfigure*

Invoke the editor to force it to read the configuration files.

\# *#load "ls1.c"*

Read in and compile the *Instant-C* library. (Skip this and the next step if your goal is to build a version without a built-in library.)

_ioinit()
Initialize the library control structures. (This should result in the printing of a four digit decimal number.)

*use **
Switch from LS1.C to the initial unnamed memory file.

#savemod ic
Write out a customized version of *Instant-C*. (You might want to write out the new copy of *Instant-C* with a different name until you have tested it.)

quit
All done. Now you can copy your IC.EXE or IC.CMD to whatever disk you wish (subject to the license agreement, of course!).

E.5 Modifying Your *Instant-C*

You may wish to customize or alter *Instant-C*. For example, you may wish to use different keyboard key assignments, or you may need to specify different screen output for your display terminal (see "Configuring the Keyboard" and "Configuring Screen Output" below). You may need to add or modify library functions for compatibility or convenience.

If you wish to simply update a version of *Instant-C* (to change some default setting, for example), the process is much simpler:

A>*ic* Start up your version of *Instant-C*.

(Make your modifications here)
You could edit a library function, or change some system variables, add more functions to the built-in library, or read in a new editor configuration

file.

\# *#savemod ic*
Write out a new version of *Instant-C*. (You may want to write it under a different name, if you have the disk space.)

\# *quit*
All done. Now you can test your new version and move it to the appropriate place for commands on your system.

E.6 Configuring a New ED

You can also configure a new stand-alone editor, ED, by a similar process. First, you should create the screen and keyboard configuration files as detailed in the above sections. Then, issue the command:

```
A>ed @
```

The editor will read the configuration files, and write a new version of itself back to disk with the name CONFIGED.EXE or CONFIGED.CMD. You can now install CONFIGED as your ED. (You should only do this after testing the editor to be sure that you have created configuration files that work.)

E.7 Changing Interrupts

Instant-C uses several software interrupts to implement some of its features. Currently three interrupts are used, although we expect to use several more (up to 15), especially for debugging support. The interrupts currently used are numbered 192 (hex 0xC0) to 199 (hex 0xC7). The reserved range is 192 to 207 (hex 0xC0 to 0xCF). Although all the interrupts from 64 on are supposed to be available to user programs, some operating systems or ROM bios codes use these interrupts. (CP/M-86 uses interrupt 224, and the ROM bios for the Heath/Zenith Z-100 appears to use the space for interrupt pointers 240 to 255 as data pointers.)

To accomodate these various systems, *Instant-C* can change the interrupt block that it uses. The system variable **_intnumber** contains the first interrupt of the block of 16 used by *Instant-C* and can be changed by setting it to some other number. After changing #intnumber and before doing any other command, you must give the **#resetint** command.

For example, to switch *Instant-C* to use interrupts 80 to 95 (hex 0x50 to 0x5F):

```
# _intnumber = 80;
# #resetint
```

WARNING: Failure to execute the **#resetint** command after changing **#intnumber** can cause your system to lock up or to fail in other mysterious ways.

After changing the interrupt block used, you can save a new copy of *Instant-C* with the **#savemod** command.

E.8 Making the Library Smaller

STDIO.H has two #defines, **_incl_float** and **_incl_scanf**, that control the inclusion of certain support code in the library. _incl_float, if #defined non-zero, will include support for floating point formatted IO in the **printf** and **scanf** functions. _incl_scanf, if #defined non-zero, will include the code for **scanf.** If you find that you run out of code space in *Instant-C* you can edit STDIO.H and #define one or both of these to zero and recover the code space. After determining which of these services you don't need, edit STDIO.H, and rebuild as indicated in Appendix E.4, above.

Appendix F

Editor Keyboard Functions

This is a listing of all the editor functions that are available, with details on how they may be used. Not all functions need to be configured; many are just slight variations of other functions.

Each function may be referenced by more than one key or key sequence. Functions with arguments (41 execute (command), for example) may be referenced by several keys, and each key can invoke the function with a different argument. So, control-W could be mapped to 41('W') to be a single keystroke 'Write buffer to file' function, and control-R could be mapped to 41('R') as a single keystroke 'Read file into buffer' function. Another example, function key F1 could be mapped to 16(12) to provide a 'half-page up' function, and function key shift-F1 could be mapped to 16(24) to be a 'full-page up' function.

The key assignment configuration program (KEYBOARD, see Appendix E) is used to change the key functions available to you.

F.1 Functions to Move the Cursor

3 cursor to beginning
Move cursor to beginning of buffer.

4 cursor to end
Move cursor to end of buffer.

5 cursor begin/end
Move cursor to beginning of buffer, or,

if already there, move cursor to end of buffer. This is a single function that can substitute for the cursor to beginning and cursor to end functions, above.

6 cursor left
Move the cursor left by one character. If the cursor is at the beginning of a line, it moves to the end of the preceding line.

7 cursor right
Move the cursor right by one character. If the cursor is at the end of a line, it moves to the beginning of the next line.

8 cursor up
Move the cursor to the beginning of the previous line in the buffer. You may find function 20, cursor to beginning of line, slightly more intuitive in result.

9 cursor up vertical
Move the cursor to the same column in the previous line. If the previous line is too short, the cursor is placed at the end of line. If the previous line has no addressable character at that column (because of a tab, for example), the cursor is set to the next character.

10 insert below, indent
Insert a line below the current line, and copy the indentation from the current line. Cursor is placed at the end of the new line (is indented).

11 cursor down
Move the cursor to the beginning of the next line.

12 cursor down vertical
Move the cursor to the same column of the next line. If the next line is too short, the cursor is placed at the end

of line. If the next line has no addressable character at that column (because of a tab, for example), the cursor is set to the next character.

13 cursor left word
Move the cursor to the left, to the beginning of a 'word' or token.

14 cursor right word
Move the cursor to the right, to the beginning of the next word or token.

16 cursor up (decimal # lines)
Move the cursor up by the number of lines specified in the argument, and set to the beginning of the line. (You are queried for the number of lines when binding a key to this function in the KEYBOARD program.) This function is useful for looking at the previous page or screen of text.

18 cursor down (decimal # lines)
Move the cursor down by the number of lines specified during configuration.

17 page up (decimal # lines)
Similar to cursor up multiple lines function, above, but takes into account the cursor centering algorithm used for the display. If the cursor is currently below the center of the screen, it is moved to the beginning of the center line of the screen. If the cursor is at or above the center line, it is moved up by the number of lines specified. Depending on the number of lines used, the page up and page down functions provide more consistent scrolling behavior. Also, for numbers of lines less than 25, you are guaranteed that every line will be displayed (not lost due to cursor centering) while scrolling through a buffer.

19 page down (decimal # lines)
Like the page up function above, but

goes down through the buffer.

20 cursor begin line

Move cursor to beginning of the current line. If the cursor is already at the beginning of the current line, move to the beginning of the previous line.

21 cursor end line

Move cursor to the end of the current line. If the cursor is alread at the end of the current line, move to the end of the next line.

F.2 Functions to Delete Text

22 delete character

Deletes the character at the current cursor position. All characters to the right of the cursor are moved left by one position. If the character deleted is the end of a line (the carriage return), the next line is joined to the current line, and all lines below are brought up by one line.

23 delete left character

Deletes the character to the left of the current cursor position. The cursor, as well as any characters to the right of the cursor, will be moved to the left by one position. If the cursor was at the beginning of a line (that is, the left character is the carriage return of the previous line), the return is deleted, the current line is joined to the previous line, and all lines below are moved up by one.

24 delete line

Deletes the entire line. All lines below the current line are moved up by one. The cursor is positioned at the beginning of the first line moved up.

25 delete to end line

Deletes all characters to the right of the cursor on the current line. If the cursor is at the first position of the line, the entire line is deleted as in the delete line function, above.

37 delete word left

Deletes from the current cursor position to the beginning of the 'word' or token to the left of the cursor. Characters to the right of the cursor, and the cursor itself, are moved left by the number of characters deleted. If the end of the previous line is deleted, the current line is joined to the previous line (except for the characters deleted).

38 delete word right

Deletes characters from the current cursor position to the beginning of the next 'word' or token. Characters to the right of the cursor are moved left by the number of characters deleted, and, if the end of line is deleted, the next line is joined to the current line.

35 un-delete

This function is very useful. It will restore, at the current cursor position, any characters, lines, or words deleted by the delete functions listed above.

When a deletion is performed, the characters are placed on a stack of deletions (in last-in-first-out order). The type of deletion is remembered, so that characters, lines, or words are properly replicated when restored with the un-delete function. The deletions stack is limited to about one thousand characters, which includes about ten bytes of overhead for each object (line, character, or word) deleted. Un-delete is useful for recovering from mistaken deletions, or as a very quick way to move text from one place to another, even between different buffers.

43 erase arg/(function)

This is hard to explain but easy-to-use function. The idea is to allow you to have the same key work differently in COMMAND mode than in an input mode.

If the editor is in command mode, this function will delete the last character of a command argument (such as a file name or search target string). When binding a key to this function during configuration, you must specify another function number which is to be executed if the key is entered during input mode. Thus, the same key can be used to erase arguments in command mode, and to perform another function (23 delete left char is recommended) when in input mode.

So, binding the Backspace key to function 43(23) will always delete the character to the left. In input or overtype mode, the character to the left of the cursor in the buffer will be deleted (function 23); in command mode, the last character of the command argument is deleted. Note: some keyboards send the same code (control-H) when you hit the left arrow key as when you hit the Backspace key; in this case, you will <u>not</u> be able to control the cursor with left arrow <u>and</u> erase with the Backspace key.

F.3 Input Mode

Text typed to the editor can be treated as either data, and stored into the buffer (Input mode), or treated as commands (Command mode). Within input mode, characters may either overtype existing characters, or they may be inserted into existing text. These functions control the selection of Insert mode or Overtype mode.

29 insert characters mode

This function selects Insert mode for text input. The text character appears at the cursor, and the cursor and any characters on the line to the right of the cursor, are moved right by one position. Inserting a carriage causes the line to be split into two lines at the carriage return. The cursor is set at the beginning of the new line, and all subsequent lines are moved down by one line in the buffer.

30 overtype characters mode

This function selects Overtype mode. In this mode, a text character entered will replace the character at the cursor. The cursor is then moved one position to the right. There are two special cases. When overtyping a tab, the tab is first replaced by the proper number of spaces, and the first replacement space is overtyped (this preserves alignment of columns, for example). The last character of a line (the carriage return), is not overtyped, but rather the line is extended by any text entered.

39 toggle mode

Go to Overtype mode, or if already in Overtype mode, go to insert mode. This allows a single key to perform mode selection.

27 insert a line

A new line is created and inserted before the current line. The current and all subsequent lines are moved down, and the cursor is placed at the beginning of the new line.

28 insert a blank

A blank is inserted at the cursor, and all other characters on the line are moved to the right by one position. This is useful in overtype mode to create space on a line without having to switch to insert mode.

F.4 Text Blocks Management

Sections or blocks of text within a buffer can be identified by setting a tag to mark one end of a block, and there are several functions for moving the block of text to and from the temporary buffer.

31 tag set Sets a 'tag' to mark the current cursor location as one end of a text block. There is only one tag per buffer.

32 tag swap with cursor
The cursor is placed at the location of the current tag, and the tag is set to the former cursor location. This allows you to see where the tag is set, and also is a quick way to switch between a place holder in the text and the current cursor location (two tag swaps leaves you where you started).

33 save block of text
The text between the cursor and the tag is moved to the TEMP buffer, replacing any previous contents. The block of text is removed from the current buffer. This a quick way to delete a large portion of text. Text in the temporary buffer can be edited separately, or retrieved to any place in another buffer. A save block function cannot be performed in the temporary buffer.

34 retrieve block of text
The text in the temporary buffer (TEMP) is inserted in the current buffer before the cursor location. It is treated exactly as if the saved text had been entered verbatim in insert mode from the keyboard.

Retrieving text does not remove it from the temporary buffer, so it can be retrieved many times. Combinations of saving and retrieving blocks of text can be used to delete, copy, or move text ("cut and paste") in a very general way. Text does not have to be retrieved to the buffer from which it was saved, so it is easy to move text within a buffer, or between buffers/files. (See the Buffer command for details on the use and management of multiple buffers.) Saved text is only modified by a subsequent save function or by direct editing in the temporary buffer. In *Instant-C*, the temporary buffer is undisturbed between editor invocations within an *Instant-C* session.

F.5 Editor Commands

46 next char is command
: This function switches the editor to Command mode, so that one of the commands can be entered.

42 cancel command/reset
: The current command is canceled without being executed. A general reset is performed: if a command or function has resulted in an error condition (search target not found, for example), the error message is removed, and further input or commands are allowed. The editor returns to the proper input mode (insert or overtype). If not in command mode, the screen is redrawn (useful for ASCII terminals to update the screen in the event of communications error).

43 erase arg/(function)
: If the editor is in command mode, you use this function to delete characters as you type in arguments (such as file names or search target strings). When binding a key to this function during configuration, you can specify another function which is to be executed if the

key is used during input mode. Thus, the same key can be used to erase arguments in command mode, or perform another function (23 delete left char is recommended) when in input mode.

41 execute (command)

This is a function that executes a particular editor command. During configuration with the KEYBOARD utility, specify an editor command letter as the argument to this function. For example, function 41(S) can set up a search command with a single keystroke. Otherwise, it is necessary to use function 46 (next character is command) and then 'S' to initiate a string search.

Note: different keys can be bound to function 41, and each key can have a different argument value. Thus, any set of the editor commands can be invoked as key functions.

47 re-execute command

This convenient function serves several roles. It indicates that execution of a command should begin, or that the command should be re-executed if already executed. You can also use this function to indicate that the command argument is complete. For example, the Search command requires that the string to be searched for be entered. The key bound to function 47 can be hit in place of carriage return to terminate the search string and initiate the command.

48 quit, save/compile

Execute the 'F'ile or 'F'inish command. Use this function to save the file (in the standalone editor) or to save and compile the function (in *Instant-C*). In *Instant-C*, upon error-free completion of the compilation, editor will return to the interpreter. This is equivalent to function 41 with an argument of 'F'.

49 quit, discard buffer
Execute the 'Q'uit command. The editor will ask for a verification, and, if affirmative, will return to the operating system (in the standalone editor), or to *Instant-C* without writing the buffer to disk or saving and compiling the function in the buffer.

F.6 Miscellaneous Functions

19 make next character control
Use this function to enter control characters. The next character entered after this function is invoked will be converted to a control character ('h' is converted to control-H, for example, and displayed as '^H'). This allows control characters to be placed into the buffer without interpretation as a key function.

50 next char is literal
The character entered after this function is placed into the text buffer directly, without interpretation as a key function and without any other translation. As opposed to function 19 (make next char control), the next character must be entered literally as a control character.

26 swap two previous characters
A common typing error (transposition) can be easily corrected with this function. The two characters preceding the cursor on the current line are reversed. Nothing happens if the cursor is at column one or two on a line. The cursor does not move after this operation.

36 swap case of character
The character at the current cursor

location is converted to uppercase if it is lowercase, or to lowercase if already uppercase. The cursor moves to the right at the end of the operation, so it is easy to change the case of a long string of characters.

40 redraw screen

The screen is cleared and redrawn.

44 translate to (char)

The argument of this function (char), is treated as input. Useful in keyboard remapping.

45 no operation

The key bound to this function will have no effect, and no text is entered into the buffer. This may be useful if you are used to a different editor, where a key performs a function different or unavailable in the *Instant-C* editor.

2 defined error

An editor error message is displayed when the key(s) invoking this function are entered. This may be useful as a reminder that some common key sequence used by another editor is not available or works differently in the *Instant-C* editor.

Appendix G

Known Bugs and Problems

This chapter lists all of our known bugs, together with any suggested workarounds. It may disappear after the field-test period is completed.

1. **#define**'s are limited to expressions with a constant value. You can't **#define** one symbol to another. (Unless the second symbol has been **#defined** to be a constant.)

We expect to fix this problem in the next version.

Also, we are working diligently in the following areas:

1. Completing full language support
2. Enhancing the library functions.
3. Finishing the documentation (especially the index).

Appendix H

Reporting Problems and/or Suggestions

This appendix details how to report bugs, and how to get credit for a new update by completely reporting a problem, or by being the first person to suggest an enhancement.

For field-test users, please send hard copy and/or a disk copy of the files that cause the problems to:

Rational Systems, Inc.
P.O. Box 480
Natick, MA 01760

or call us at (617) 653-6194.

We will try to fix all problems as quickly as possible and get a new version to you within a few days.

User's Manual

Instant-C™

Index

User's Manual

Instant-C™

Instant-C™

User's Manual