Volume 2 Number 4 Price $6.95

THE C JOURNAL™

Sources and Resources for All C Programmers Spring 1987

Free-Standing C
(or, Living Without an Operating System)

Microcontrollers
Case Studies
Using yacc

Embedded System Programming Considerations

Ken Milnes and George Loughmiller
Etak, Inc.

Introduction

C is a good choice of a high level language in an embedded system. It allows the programmer enough flexibility such that little assembly code should be required. Modern compilers are available which generate fast and compact executable code, further reducing the requirements for assembly code.

C was successfully used to develop the software for the Etak NAVIGATOR™. The NAVIGATOR is a vehicle navigation computer which displays a digital road map on a vector display in a passenger vehicle and navigates along the road network to position the vehicle at the correct place on the map. Additionally, a destination may be entered by street address, and the NAVIGATOR will display a star at the destination location. All of the software was written in C and assembly code using an IBM PC/AT running PCDOS as the development computer. The NAVIGATOR uses an Intel 8088, with 256 KB of random access memory (RAM) and 16 KB of read only memory (ROM). A cassette tape is used to store the navigation program and the map database.

The choice of the development environment when writing programs for an embedded system is one of the most important decisions to make. Developing the programs on a computer which uses the same instruction set as does the embedded system will reduce the amount of time required to debug the system. By using the same instruction set, much of the executable code may be debugged on the host development computer where the debugging tools are generally superior. In many cases the actual application program may be run on the host development computer in real-time. After choosing the host development computer, a C compiler must be selected. Most compilers are designed to generate code which will run in the host computers environment. To port this code to an embedded system, some modifications will be required to the C startup functions to account for the differences between the host and target environment. Some compilers are designed specifically to generate programs for an embedded system. These compilers do not make assumptions about the target environment and will supply documentation on what is required by the startup code to execute the C code.

The Etak Navigator software was developed using a compiler which was designed to run under PCDOS. This facilitated efficient debugging because of the high-level tools available such as symbolic debuggers and file and screen I/O systems. All of the program modules which were not specific to the embedded system could be debugged using these tools without having to execute the programs on the embedded system. By using this type of compiler, some special problems had to be solved. A program had to be written to convert the executable code to hexadecimal

† NAVIGATOR is a trademark of Etak, Inc.

such that a ROM can be programmed. Little documentation was provided by the compiler vendor on what was required to initialize the system before the C code could be executed.

A discussion of the differences between the environment of a host operating system and an embedded system follows:

Startup code

Before the main function of a C program is executed, the operating environment of the computer must be initialized. A routine must be written to do this initialization and then call the function `main`. After `main` returns, (if it does), a routine must be provided for the C program to return to. The functions of the startup code may include:

1. Establish a stack for the program.
2. Zero uninitialized data.
3. Establish and initialize the heap.
4. Check memory requirements.
5. Setup segment registers (for Intel 8088/8086)
6. Command line processing.
7. Open `stdin`, `stdout`, and `stderr`
8. Initialized any special hardware such as a floating point processor.
9. Call to function `main`.
10. Exit from `main`.
11. Close any open files on exit

In an embedded system, many of the above items may not be required. For example command line arguments most likely will not be used. To rewrite the startup code, it will be useful if the source code of the compilers startup code is available as a starting point.

Establish a stack for the program: Various compilers and compiler models place the stack in different places. The particular environment in the embedded system may place some constraints on the placement of the stack. Care must be taken to maintain compatibility with any stack checking which the compiler does at the start of each function or to disable the checking via a compiler switch.

Zero uninitialized data: All uninitialized data is normally zeroed before the main program is executed. The location of these areas may be determined by examining the load map and deducing the segment names or by example in the compilers startup code or documentation.

Establish and initialize the Heap: If the heap is used by the program (calls to `malloc`, `calloc` use the heap), it must be established in the startup code. In some cases this is done by initializing some global variables which define the start of memory where the heap will reside. The startup code for the compiler will provide an example of how to initialize the heap.

Check memory requirements: Before the main program is executed by the startup code, a check is made to insure that enough memory exists for the program to successfully run. This check may not be required with an embedded system.

Setup segment registers: Depending on the compiler and compiler model used, the segment registers (DS, ES, SS) must be initialized. This is the responsibility of the startup code. See the documentation for the particular compiler on what these variables must be set to.

Command line processing: If a command line is appropriate for the embedded system, the variable `argc` and array pointed to by `argv` must be initialized, and then passed as arguments to `main`.

Open `stdin`, `stdout`, and `stderr`: If these devices exist in the embedded system, they must be opened before `main` is called.

Initialize any special hardware such as a floating point processor: This initialization may be done in the startup code or by the boot code of the computer.

Call to `main`: This is the point where the user's main program is executed.

Exit from `main`: If the main program ever returns, an exit routine must be provided. This routine follows the call to `main`.

Close any open files on exit: If a file system exists, then exit normally will close any files which were left open.

Segment Relocation

When a host computer executes a program, the executable code is retrieved from a disk file and placed in memory. In many cases, the code will require segment relocation. That is to say, some of the address information must be modified based on the loaded programs location. Linkers generate a fixup table which specify the offsets into the load module where relocation is required. In the case of code which is in read-only memory (ROM), the fixups must be applied to the program before the ROM is programmed. If the program is going to reside in random access memory (RAM), then the fixups may be applied before or after it is loaded into memory. If the memory address where the program will reside can not conveniently be the same every time, then it is necessary to apply the fixups after the the program is loaded into memory and the location of the code is known.

A program fixup is generated by the linker whenever a call to an absolute memory address is encountered or when a reference to a code or data segment is made. The fixup table is a list of pointers into the program which reference absolute memory addresses. The linker generates op codes with the assumption that the program will be loaded at absolute memory address 0000H.

When the base address of the program is known, then this value is added to the values pointed to by the fixup table.

The following listing (*Listing 1*) is a code fragment which will use an MSDOS fixup table to apply fixups to an unmodified EXE file. See the DOS Technical Reference manual for details on how a DOS EXE file is laid out and the structure of the header attached to an EXE file.

Read Only Memories

The major consideration which must be kept in mind when writing programs to run in ROM is the data variables. Programs normally have initialized and un-initialized data areas. The initialized data will be included in the EXE file and must be programmed into the ROM along with the code. The startup code in the ROM must move this initialized data to the memory address where the data will eventually reside. This data area is normally at the beginning of the data segment, but does not have to be. A review of the linker's load map will reveal where it resides for a particular program.

C programs do not normally modify any of the code during run time. If it is required to do this for some reason, the code must be moved to RAM for obvious reasons.

C Library Routines

C provides many convenient I/O services such as file I/O and screen I/O. In many embedded systems these are not required. The C file and screen I/O can take up as much as 5 to 10 KB of code, so it is desirable to delete these routines from the executable code. In principle, it should be easy to delete these routines by not making any calls to the functions which do the I/O (`read`, `write`, `printf`, `scanf`, etc.). It is often not as simple as this. Some compilers include a label in the object modules to force the DOS startup code to be included at link time, which will force much of the I/O system to be included. For example, the Microsoft C compiler includes the label `_acrtused`. Microsoft declares `_acrtused` in the startup code `_astart`. By simply declaring this label elsewhere in your code, the Microsoft startup code will not be required.

The Computer Innovations C86 compiler will force much of the I/O system to be included if `stdio.h` is included in any of the modules.

If the compiler vendor supplies the source code for the run time libraries, it is possible to customize it to suit the particular application. For example the `sprintf` and `sscanf` functions are useful even if an I/O system is not present. In many cases floating point is not required by the applications. This code may be deleted if it is not required and will save substantial memory. If `sprintf` or `sscanf` are used, they must be modified to not use floating point.

```
/* this function will use the fixup table to modify the op-codes
   in a PCDOS .exe load module.  The following assumptions have
   been made :
   1) Output of the linker has not been modified.
   2) This function compiled under large model (pfixup must
      be a far pointer).
*/

relocate(fixup,codebase,numfixup)
short baseaddr;                /* segment address of
                                  beginning of load module */

short *fixup[];                /* array of fixup entries */
short numfixup;                /* size of fixup array */
{
short *pfixup;

   pfixup = fixup;
   while(numfixup--) {
      *pfixup++ += baseaddr;
   }
}
```

One useful trick for resolving link problems where too much of the C library is being pulled in, is to delete from the compiler's run time library the functions which are not desired. The linker will then specify which labels are unresolved and which functions are requesting them. This should lead to the problem's identification.

Debugging

One of the most important aspects to consider in an embedded system is debugging of the executable code on the target hardware. The availability of terminals, printers and powerful debugging tools such as symbolic debuggers and dumping programs greatly accelerates debugging on the host development computer. However, because of the inherent differences between the host environment and that of the target, not all bugs can be caught at the host level, and some debugging must take place on the target hardware. With the proper design of both hardware and software, the amount of time spent debugging on the target system can be minimized.

As previously discussed, choosing a development computer which has the same type of processor as the target system, simplifies debugging in that the actual compiled/assembled code is debugged on the development computer. Additional steps can also be taken on the host computer to simplify debugging. For instance, it is desirable to interface the target peripherals to the host computer. In the Etak system, an interface board for the IBM PC was developed for the vector display device, and most of the development and debugging of the display driver and map display software was completed prior to the availability of the target hardware. Other peripherals can be mimicked by the resources available on the host computer. For example, the Etak database is stored on cassettes; however, for program development on the host computer the database is stored on the hard disk. Requests for database on the development computer are simply redirected to disk.

Efforts should be made to keep the differences between the host and target versions at the lowest levels of the software. In this manner, the number of modules which differ from host to target versions can be minimized and confined to special libraries, those for the development computer and analogous libraries for the target hardware. To switch from a host to a target version of the program is then simply a matter of linking with a different link file which includes the appropriate libraries.

Whereas the above steps can minimize the time spent debugging on the target, bugs will crop up when the program is ported to the target hardware, and some debugging must take place on the target hardware. One of the most powerful tools which can be used in this task is an in-circuit emulator (ICE) which replaces the processor giving the user access to the CPU registers and memory while allowing the system to run at its normal speed. An ICE allows the user to set conditional breakpoints, examine and modify program memory as well as data, and trace program execution and memory address contents. In addition, code can be disassembled and the program may be altered by assembling directly to the target memory. Many of the emulators allow symbolic addressing and still others allow debugging at the source language level.

While the use of an emulator greatly facilitates debugging of the executable code on the target hardware, it does however have several drawbacks. The high cost of an emulator makes it prohibitively expensive to install an emulator in more than a few systems, therefore, an alternate means of debugging on the target hardware must be available. In a system as complicated as the Etak Navigator, some bugs have a mean time between occurrence of hundreds of hours or thousands of miles in the case of a vehicle computer. This necessitates some form of a resident debugger.

In order to debug on the target system without the aid of an emulator, some sort of interactive display device must be available. This can be something as simple as a front panel display register, though a monitor is preferable. If the target hardware does not include a monitor it may be possible to design the hardware such that a monitor can be added to selected units. Once a monitor is available, a simple debugger can be quickly added to the executable code allowing the user to interactively examine and modify memory. In this manner, bugs which occur infrequently can be captured and analyzed without relying on an ICE. More sophisticated debugging tools such as breakpoints, dump routines, data checkers, etc. can also be implemented for analyzing problems and performance. An asynchronous serial communications adapter is very useful to provide communications between the host and target computers. This will allow downloading of test programs and dumping memory images from the target computer.

Inevitably, a complicated piece of software will contain fatal bugs, i.e. problems which require the system be reset through intervention of the hardware. To accomplish this, the hardware must monitor the activity of the software and assert the reset line on the processor when things are not in order. In the Etak system, the hardware monitors write commands to an I/O port and reset the processor when no activity has occurred for a specified amount of time. An analogous software function, known as watchdog code (see "Software fault tolerance staves off the errors that besiege microprocessor systems", Dick Jarrett, *Electronic Design*, August 9, 1984) can be implemented to enhance the hardware function. The watchdog code is set up to test not only the time interval spent in a section of code, but also the sequencing of the code. The error detection is done by monitoring the activity of various parts of the main program and all of the interrupt routines. If a sequence error or loop timing error does occur, then the software will stop writing to the special I/O port, thus causing the hardware to activate the reset line.

By integrating the on-board debugger into the watchdog code such that the debugger is automatically called up when a watchdog problem is detected, an extremely valuable tool is available for trapping these fatal bugs. In the Etak system, the interrupt routine which monitors the various watchdogs freezes as much of the system as possible and immediately calls up the debugger. In this manner a postmortem can be performed on the failed code. The values for all of the registers and flags at the time of the interrupt are displayed so that the routine which was executing at the time of the interrupt can be determined. The stack can be examined so the local variables can be determined and the sequence of function calls can be traced back.

The time spent in developing onboard debugging tools on the target hardware is paid back many fold by accelerating up the debugging process. The original coding for the Etak debugger was accomplished in less than a week. The watchdog trap feature enabled the capture and analysis of fatal bugs which would have taken months of driving in an emulator-equipped system.

Ken Milnes is a senior engineer at Etak, Inc. He is responsible for navigation, communications, and cassette operating system software development in the NAVIGATOR.

George Loughmiller is a senior engineer at Etak, Inc. He is responsible for navigation, display, database and debugger software development in the NAVIGATOR.

∞

Next Issue

- A surprise SPOT_LIGHT guest
- Tom Plum's Report on the March '87 ANSI X3J11 meeting in Boulder, Colorado. This will be the first meeting after the end of the 4-month public review period of the Standard.
- Several articles on communications including CRC, X.25 and CCITT
- Software engineering and coding standards, including portability considerations
- More Words of wisdom from Doctor C
- Tips from our UNIX system gurus
- Our regular columns

∞