Dr. Dobb's Software Tools for the Professional Programmer

Retargetable Concurrent Small C

Dr. Dobb's Journal August 1997

Porting CSC to the 8051

By Andy Yuen

Andy is a consultant working for the Professional Services Organization of Hewlett-Packard in Sydney, Australia. He can be reached at andyyuen@ozemail.com.au.

Since presenting "Concurrent Small C" in *DDJ* (August 1996), I have been inundated with e-mail asking when the 8051 version will be available. The apparent demand for an 8051 version is probably due to CSC's language-level support of preemptive multitasking, synchronization, and interrupt handling, and partially to the lack of a freely available C compiler for the 8051. The so-called "freeware" C compilers, to my knowledge, limit the size of the generated code to 1 or 2 KB, which make them practically useless, even for hobbyist projects.

In this article, I'll present a version of CSC that is retargetable, with the port available here targeting the 8051. I've selected the 8051 for two reasons:

- The 8051 is one of the world's most popular 8-bit microcontrollers: Over 120 millions units (including variants) were sold in 1993 alone.
- The 8051 poses a special challenge to a compiler writer because of its peculiar architecture and nonorthogonal instruction set.

The Retargetable CSC (RCSC) distribution (source code and related files and tools) is available electronically; see "Availability," page 3.

Concurrent Small C Organization

To make CSC easily retargetable, you have to separate the code generator from the compiler itself. Otherwise, for each port, you have to modify the compiler -- not an easy task if you are unfamiliar with CSC internals.

CSC, like Small C Version 2.2, consists of four modules:

- cc1.c handles the overall program flow and parsing.
- cc2.c handles the input preprocessing such as macro expansion.
- cc3.c handles expression analysis.
- cc4.c handles code generation and optimization.

CSC generates intermediate codes in memory called PCODEs (P for pseudo), which are instructions for a 2-accumulator, 16-bit virtual machine with a stack pointer. It then runs the PCODEs through a peep-hole optimizer before translating them into the target processor's assembler language using a translation table contained in

cc4.c. There are a total of 109 PCODEs defined. A brief description of PCODEs and their addressing modes can be found in the header file cc.h. For an in-depth discussion on PCODEs, see James E. Hendrix's *A Small C Compiler* (ISBN 0-13-814724-8), or *Dr. Dobb's Small-C Compiler Resource CD*.

RCSC outputs PCODEs instead of target assembler code. A macro processor translates the PCODEs into the target processor's assembler code; see Figure 1. The macro processor must possess features such as arithmetic capabilities, conditional macro expansion, and string manipulation (a task that the GNU M4 macro processor handles deftly). RCSC requires that a set of macros be defined for each target machine to implement the PCODEs. The macro processor takes the intermediate PCODE file generated by the compiler as input and processes them using the set of processor-specific macros to generate the target machine's assembler code. The advantage of this approach is that a program needs to be compiled only once. The macro processor takes the same RCSC-produced PCODE file as input and produces code for different processors using different macro sets. The resulting file is then converted into executable format using the target platform's assembler, linker, and run-time libraries. Since the PCODE has been optimized by the compiler, the translated assembler code for different processors is also optimized.

8051-Specific Design Issues

The 8051 has an unusual architecture that separates memory into five different areas: code (ROM), internal RAM, bit-addressable memory (a certain area of the internal RAM), special function registers, and external RAM. It uses different instructions to access different memory areas. The 8051 has an unusually nonorthogonal instruction set. For example, it uses the DPTR pointer to access external RAM and code segments. It has an INC DPTR instruction to increment the pointer but no corresponding decrement instruction, which makes address manipulation a bit inconvenient. Also, the 8051 has an 8-bit hardware stack pointer, which means that the maximum possible stack size is 256 bytes, hardly enough for reentrant functions.

To get around the small stack limitation, 8051 C compilers usually implement their own stacks in software. As this will impact performance to a certain extent, most compilers have at least three different programming models to cater to different needs: small, medium, and large. The small model does not support external RAM access. All the program's data must fit in the 8051's internal RAM. All function parameters are passed using registers. The medium model supports external RAM access and uses either the 8051's paged memory access (for example, accessing external memory using the P0 and P2 ports and indirect addressing mode) or registers to pass function arguments. Both models are nonreentrant, but avoid the performance degradation associated with a software-stack implementation. The large model supports external RAM access. It also implements a software stack to pass function arguments. Some implementations do not support reentrancy by default; users have to use pragmas to enable it explicitly.

If all the program's data can be squeezed into the 8051's 128-byte (256 bytes for 8052) internal memory, you probably don't need RCSC's multitasking features. So, the small memory model can be eliminated. By definition, RCSC requires reentrancy for concurrent operation; therefore the nonreentrant medium model is obviously not acceptable either. Consequently, RCSC only provides one programming model -- the

large reentrant model -- which implies that it implements its own software stack. RCSC always generates reentrant code. The downside is that reentrant code is slower than the two nonreentrant models because of the software-stack operations.

Implementation

I modified cc1.c and cc4.c to generate PCODEs instead of target-specific assembler code. A number of pseudo PCODEs have been added to give more information to the code generator, but these pseudo PCODEs usually do not themselves cause code to be generated. The closest analogy to a pseudo PCODE is an assembler directive. Table 1 summarizes these new pseudo PCODEs and their intended use.

The GNU M4 macro processor is used for expanding the PCODEs. It reads text one line at a time from the input stream and scans the text for macros that have been defined. When M4 finds a macro, it replaces it with the macro definition and puts the replacement text back to the input stream. It then reads another line of text from the input stream and repeats the process until there is no more input. A macro expansion is invoked either by a macro name or a macro name immediately followed by a number of arguments like a C function call; see Example 1(a). A macro is defined using the built-in define macro as in Example 1(b), where *OPxy* is the macro name and what follows is the macro definition. M4 replaces the string *$n* with the value of the *n*th argument. A macro call like Example 2(a) results in Example 2(b). *ifelse* and *len* are M4 built-in macros. *ifelse* compares the first parameter with the second one. If they are the same, the value returned is that of the third parameter. If not, and if there are more than four arguments, the process is repeated for arguments 4, 5, and 6; otherwise, the value returned is that of the last argument. *len* returns the length of a string. M4 contains many more built-in macros and other powerful features not shown here. (For more information, see the documentation that comes with the M4 package.)

A set of M4 Macros (like that above) is defined to expand PCODE into processor-specific assembler code. The macros for the 80x86 are straightforward, most of them are copied from the original translation table contained in cc4.c with minor modifications. The macros for the 8051 are more complicated because the 8051 is an 8-bit processor and the PCODEs are instructions for a 16-bit virtual machine. Consequently, many more 8051 instructions are needed to implement the PCODEs than the 16-bit 80x86 instructions. The macro definition files are 8086.M4 and 8051.M4 for the respective processors. Listings One through Four resent the C code, RCSC-generated PCODE, and M4-generated assembler-code listings for the 80x86 and 8051 for the *strcpy* function.

Again, the virtual machine consists of three 16-bit registers: the primary and secondary accumulators, and the stack pointer. The register usage for the 80x86 and 8051 are summarized in Table 2. The 80x86 uses the hardware stack, which grows toward low memory, while the 8051 uses hardware and software stacks, which grow toward high memory. Such a choice for the 8051 is to simplify variable access since the 8051 assembler directive DW stores the high byte before the low byte (Big-endian).

A stack frame is created on every C function invocation by the function prolog ENTER and released by RETURN. Figure 2 shows the differences between the two stack-frame implementations. Assembler programmers must fully understand the stack frame structure to interface assembler routines successfully with RCSC programs.

Figure 3 shows fragments of assembler code to access function parameters and local variables allocated on the software stack within a C function. If you understand the 80x86 stack frame setup, the easy way to do it on the 8051 is to use PCODE GETw1s(*n*), where *n* is the same offset used for the 80x86, to access the function parameters and local variables. If you are using PCODEs in your assembler program to interface to RCSC functions, you have to process it using the M4 macro processor. The 8051.M4 macro will take care of the translation for you. I recommend you take the time to get familiarized with the PCODEs because they may come in handy while coding in assembler. When you are using PCODEs to call a C function, make sure you remove the function arguments from the software stack on return from the function call by using ADDSP(*n*), where *n* is the number of function arguments. This is necessary because, for C, unlike Pascal, the calling program is responsible for such a chore, not the called function.

In the 8051 implementation, whenever a C function is called, the first thing the called function does is pop the return address from the hardware stack and store it in the stack frame on the software stack. Aside from providing a recursion level limited only by the allocated size of the software stack (declared when defining a task), moving the return address from the hardware to the software stack also makes task switching quicker, because there are less data on the hardware stack to copy.

When a task is declared, the kernel allocates the amount of stack space specified, or defaults to STKSIZE bytes if the stack size is left unspecified. The initial values for all the relevant registers are initialized on the software stack. Of particular interest are the return address and the hardware- and software-stack pointers. Together, they ensure that the task will be started correctly when it runs for the first time. When a timer interrupt occurs, the current task's context, its CPU register contents, and the data on the hardware stack are copied to the task's descriptor block. Saving the contents in the hardware stack is necessary because each task owns the hardware stack when it is running due to its small hardware-limited size. This is followed by restoring the context of the task to run next. This involves copying the context saved on the task descriptor back to the cpu registers. Again, the data on the hardware stack at the time this task was last run is also restored before it resumes execution. While saving the registers always takes the same amount of time (fixed numbers of registers to save), the same is not true for the data on the hardware stack if uncontrolled. The hardware stack may be empty (best case) to full (worst case). To guarantee the task-switching time, you want to limit the amount of data stored on the hardware stack. RCSC does this in two ways: moving the return address on entry to a C function from the hardware to the software stack, and controlling the hardware stack in the run-time library. While RCSC uses the 8051 LCALL instruction to invoke a C function, the called function immediately pops the return address from the hardware stack and saves it in a stack frame created on the software stack. Consequently, the depth of the hardware stack does not increase even for recursive calls. The run-time library is mostly written in 8051 assembly code for performance reasons. As such, most of the instructions don't use the software stack. Function parameters are passed using registers. I've taken care not to exceed a call level of three. RCSC users should also observe this rule when using assembler code to interface to RCSC programs to guarantee task switch performance.

Registers starting from bank 2 are used to implement RCSC's virtual machine with the hardware stack immediately following them. My intention is to simplify context

switching. Context saving is performed by simply copying the registers starting from bank 2 to the register pointed to by the hardware stack pointer.
Like the 80x86 implementation, the compiler collects information on monitor initialization and task information so that RCSC can execute the monitor initialization code and create tasks at startup. On hardware reset, *startup* (located in kernel.c) is executed. It calls *initvar* to initialize all variables in the XDATA segment by copying the literals and constants from the code segment. The watchdog timer (if one is present) should be activated only after the copying is complete. If that is not possible, then users have to add code to reset the watchdog timer in the *initvar* routine to prevent it from expiring. Once the variables are initialized, RCSC looks at the table containing the addresses of monitor-initialization routines (built by the RCSC compiler) and executes them before allocating stack space and creating user-defined tasks whose starting addresses are in the task table. It then converts *main()* into a task, and initializes the timers and serial port and goes into multitasking mode.
The interrupt mechanism remains unchanged from CSC Version 1 -- all interrupts share a common interrupt function that saves the task context, determines the source of the interrupt, executes the user-defined interrupt function, and restores the context. On the 80x86, all interrupt information is held in a list of interrupt descriptors. On the 8051, the information is compiled into the CODE segment in Example 3.
For example, the timer interrupt vectors to *__intr_1,* which calls the common interrupt handler *__handler*. The return address on the stack points to the user-defined interrupt function. Immediately following it is the interrupt number. Hence, it is easy for *__handler* to determine the location of the user interrupt function and the interrupt source. Table 3 summarizes the interrupt sources supported by the 8051 version of RCSC.

Complications

RCSC requires a relocating assembler that supports separate assembly and linking. Unfortunately, such a tool is hard to find in the 8051 world. None of the 8051 freeware assemblers I found (with one exception) support separate assembly and linking. All source code has to be put in one file to be assembled into an Intel hex format output file. One of the packages in this category worth mentioning is W.W. Heinz's ASEM-51, which has excellent documentation and a bootstrap program.
The only freeware assembler I could find that supports separate assembly and linking is CAS, the 8051 C-Assembler written by Mark Hopkins. It is a nice assembler that possesses all the usual 8051 assembler features plus some novel nonstandard ones. The only features missing for RCSC's needs are the support of public named segments (that can be used to combine data from different modules into a contiguous area) and a librarian utility (to archive assembled object files). These features are imperative for RCSC to group all variables and task and monitor information into contiguous areas for use during initialization.
To simulate these features, I had to resort to a number of UNIX tools ported to MS-DOS, such as awk, grep, and uniq, to manipulate the RCSC output to control the placement of variables in the code and data segments. (All tools used in this project are included in the RCSC distribution.) Because of this, to compile and link an 8051 RCSC program consisting of multiple .c modules is not as straightforward as with the 80x86 version. The procedure involves the following steps:

1. Compile an RCSC module (with extension .c) into PCODE (with extension .m4).
2. Use awk to reformat the output, extract data storage information to provide the named segment effect at a later stage, and record external function references.
3. Use the M4 macro processor with the macro definition file 8051.M4 to expand the PCODE file into a file containing assembler code (with extension .s).
4 Repeat step 1 until all .c modules are processed.
5. Use awk and the information collected in step 2 to generate the header file memmap.i, containing symbols defining absolute address to control placement of code and data.
6. Assemble all .s files into object files (with extension .o).
7. Create and compile a module that contains all referenced library functions to be included for linking.
8. Link all object modules to produce an executable in Intel hex format (with extension .hex).

Fortunately, all these steps can be automated using a combination of batch files and makefiles. You may want to use the makefile for the 8051 demo program as a template for new projects. Creating an executable for the 80x86 version is much simpler, involving only steps 1, 3, 4, 6, and 8. The 80x86 executable, assembler, and object files have extensions .exe, .asm, and .obj, respectively, instead of .hex, .s, and .o for the 8051.

Step 2 is required to group all literals and constants in the CODE segment and variables in the XDATA segment in contiguous areas in memory. The requirement exists because, at startup time, RCSC calls *initvar,* which initializes all data variables in the XDATA segment by copying the literals from the code segment. Keeping them in contiguous areas in the same order facilitates the copying process. As mentioned earlier, the CAS assembler does not support named segments like the 80x86 macro assembler does. You have to control the placement of data by specifying the absolute starting address of a segment as in: SEGMENT XDATA AT 100.

The awk script groups and moves all memory-allocation statements such as DW, DB, and DS (which may scatter all over a module) to the end and keeps track of the number of bytes allocated. For example, if the current module allocated 100 bytes and the XDATA segment starts at 100 as in the previous example, the next module automatically places the data immediately following the previous module by using: SEGMENT XDATA AT 200.

The same approach is used to collect task and monitor function addresses in contiguous areas for use during system initialization. The information is kept in a file with the same name as the .c program being compiled (but without a file extension). For example, if the program module being compiled is testprog.c, then the data allocation information is kept in testprog. This means that each module making up the application creates its own information file.

This step also maintains a list of all external function references so that these functions can be included in step 7. Unlike the data placement-control information, the external references for all modules are kept in the file libc.

The placement of code and data is governed by the RCSC configuration file rcsc.cfg, which must be located in the same directory as the modules being compiled. This configuration file defines five parameters for customizing RCSC to generate code for different 8051 memory configurations:

- CLITBEG defines the starting location in the code segment where all literals and constants are to be placed. This usually points to the area right after the interrupt vectors.
- LASTROM tells RCSC the last location of the installed ROM (code segment).
- DVARBEG defines the starting location in the XDATA segment where program variables are to be placed. It must not have a value of zero because the NULL pointer in RCSC is defined as zero. If you use zero, you will not be able to access the variable placed at location zero.
- LASTRAM tells RCSC the last location of the installed RAM (XDATA segment).
- STKSIZE defines the default size of the stack for each task if the stack size is not declared in the task function. RCSC always creates a stack of this size for *main()*.

Figure 4 shows the memory organization of an RCSC program. Literals/constants, monitor/task tables, and the application code are all placed in the CODE segment. The heap starts immediately after the system and program data variables up to LASTRAM in the XDATA segment. Stacks for various tasks are allocated from the heap when they are created.

Step 5 takes all the information collected in step 2 and generates the header file memmap.i, which contains constant definitions for the absolute location for the SEGMENT XDATA AT and SEGMENT CODE AT statements described in step 2.

Step 7 is needed because there is no librarian utility to archive assembled C library object files for use in the linking process to produce an executable. Instead, step 7 uses the information on external function references collected in step 2 (in the file libc) to generate the libc.c file (which contains *#include* statements of all C library modules needed by the target), compiles and expands it into assembler code, and assembles it to produce the object file libc.o. Libc.o is then used in the linking process to produce an executable. If there is no change in libc.c, it will not be recompiled. The dependencies of the various external library function references are found by using the index file file.idx located in RCSC's LIB51 directory. It has the format *functionname:file1:file2:...:filen*, where *functionname* is the name of the function, which depends on files: *file1* through *filen*. All name fields are delimited by ":". For each file named, an include statement is added to lib.c for that file. To avoid a file being included multiple times, the generated libc.c file is fed through the *sort* | *uniq* command pipeline to remove duplicate entries. If you want to add frequently used functions to LIB51 so that they are automatically included in the linking process, you must add an entry in file.idx for each function added. One limitation is that functions included this way must not contain any global-variable declarations. Users must include kernel.o as one of the targets in the makefile because kernel.c contains the function *initvar* that handles the copying of literals and constants from the CODE to the XDATA segment at startup time. To get that information, it includes memmap.i and has to be reassembled each time memmap.i is changed. *initvar* also initializes the variables *__memptr*, *__memend*, and *__stksize* with the heap starting address, last RAM address, and the default stack size, respectively, for use by the memory-allocation routines *calloc()*, *malloc()*, and *free()*.

80x86 and 8051 Version Differences

The differences between the two implementations (aside from the way an executable is produced) arise because of their operating environment differences. The 80x86 version assumes the presence of an operating system, namely MS-DOS. The target 8051 program is the operating system itself.

The 8051 version does not support a file system. Therefore, file I/O functions such as *fopen*, *fread*, and the like are not supported. The C standard I/O functions *putchar*, *getchar*, *puts*, and *gets* provide access to the serial port to communicate with the outside world, while *sscanf* and *sprintf* handle formatted input and output. Since the 8051 operates in the embedded-system environment, there is no need to pass command-line parameters to it through: *main(argc, argv)*. Consequently, *main* does not support arguments.
Unsigned integer arithmetic is much more efficient than its signed counterparts for the 8051 because all signed multiplies and divides are first converted from two's complement form into a sign/magnitude representation before invoking the corresponding unsigned routines to carry out the operations. The result is then converted back to two's complement form. However, there is no difference in efficiency between signed and unsigned operations for the 80x86 because they are supported by the 80x86 instruction set directly.
Unlike the 80x86 version, the 8051 version does not allow the use of standard C I/O functions inside the monitor initialization code, because the serial communication has not yet been set up at that time.
Watch out for the difference in the stack implementation. The 80x86 uses the hardware stack that grows downward while the 8051 uses a software stack that grows upward. This does not cause a problem as long as you are not using variable argument lists in your functions. The problem comes about in the pointer arithmetic that is used in locating the first argument and the way the subsequent arguments are accessed; see Example 4. Although the function is called with a variable number of arguments, RCSC defines the function as having only one argument and uses pointer arithmetic to get the address to the first argument (because, unlike standard C, RCSC pushes arguments on the stack from left to right). CCARGC() is an RCSC run-time routine that returns the number of arguments passed on the stack.
The different stack implementations require users to keep two versions of the source code, which is obviously unacceptable. To overcome this problem, I created M4 macros to mimic the ANSI variable argument facility: *va_list*, *va_arg*, *va_start*, and *va_end* in the files STDARG51.M4 and STDARG86.M4. Functions with variable argument lists should be modified to use the *stdarg* macros to maintain a single source. Using *stdarg* macros also makes the program easy to understand; see Example 5.
An example can be found in the LIB51 directory, where sprintf.va is the source and sprintf.c is the implementation-dependent file generated by using the M4 macro processor and the STDARG51.M4 macro definition file using the command M4 ..\BIN\STDARG51.M4 SPRINTF.VA > SPRINTF.C.

An 8051 Example

To demonstrated the portability of RCSC, I've used the dining philosopher example used in the original CSC article, replacing the asyn8250.c module with asyn8051.c. It uses the function *puts* to output to the serial port that connects to either a PC or an asynchronous terminal. This example also demonstrates that, to keep RCSC programs portable, hardware-dependent parts (such as serial communication) should be put in separate modules.

Conclusion

Since this is the first release of the 8051 implementation, emphasis is on producing correct rather than optimized code. Consequently, interrupt handling is not as fast as I would like it to be. Also, the M4 macro definitions do not explore the use of indirect addressing modes with ports 0 and 2 to access variables placed in external RAM, which could obviate the frequent reloading of the DPTR pointer. Furthermore, the issues of task termination and exception handling are still not being addressed in this release. The procedure for creating an 8051 executable is more complicated than I would like, due to the lack of an assembler that supports named public segments, separate assembly/linking, and a librarian.

On the bright side, RCSC can be ported to new processors without modifying the compiler itself. This is achieved by separating the code-generation part from the compiler. The RCSC compiler now produces intermediate PCODE as output instead of assembler code. The M4 macro processor is used as the code generator to expand the PCODEs into target assembler code. Consequently, porting RCSC to a new processor only involves writing M4 macro definitions for the PCODEs and adapting the kernel and C libraries to the new environment. An embedded-systems engineer can now write a concurrent program in RCSC using all of its language-supported preemptive multitasking, synchronization and interrupt-handling capabilities; compile it once; and port it to different processors by simply running the code through the target-specific RCSC code generators. An RCSC concurrent program is portable if it does not use any processor-specific facilities such as pulse-width modulation, serial I/O, analog-to-digital conversion, and so forth. Since an embedded system is bound to use some of these facilities, it is a good practice to move such code into their own modules so that they can easily be replaced for a specific environment.

Listing One

```
/*  ** copy t to s  */strcpy(s, t) char *s, *t; {

  char *d;
  d = s;
  while (*s++ = *t++) ;
  return (d);
  }
```

Back to Article

Listing Two

```
TOSEG(2)CSCEXTRN
ENDSEG(2)
TOSEG(1)
ENDSEG(1)
TOSEG(2)
DECLPUBLIC(_strcpy, 2)
ENTER
ADDSP(-2)
GETw1s(6)
POINT2s(-2)
PUTwp1
LABm(_2:)
POINT2s(6)
GETw1p(0)
INCwp
PUSH1
```

```
POINT2s(4)
GETw1p(0)
INCwp
MOVE21
GETb1p(0)
POP2
PUTbp1
NE10f(_3)
JMPm(_2)
LABm(_3:)
GETw1s(-2)
RETURN(2)
ENDSEG(2)
END
```

Back to Article

Listing Three

```
CODE SEGMENT PUBLICASSUME CS:CODE, SS:DATA, DS:DATA
extrn __eq: near
extrn __ne: near
extrn __le: near
extrn __lt: near
extrn __ge: near
extrn __gt: near
extrn __ule: near
extrn __ult: near
extrn __uge: near
extrn __ugt: near
extrn __lneg: near
extrn __switch: near
CODE ENDS

 DATA SEGMENT PUBLIC
DATA ENDS
 CODE SEGMENT PUBLIC
ASSUME CS:CODE, SS:DATA, DS:DATA
PUBLIC _strcpy
_strcpy:
PUSH BP
MOV BP,SP
ADD SP,-2
MOV AX,6[BP]
LEA BX,-2[BP]
MOV [BX],AX
_2:
LEA BX,6[BP]
MOV AX,[BX]
INC WORD PTR [BX]
PUSH AX
LEA BX,4[BP]
MOV AX,[BX]
INC WORD PTR [BX]
MOV BX,AX
MOV AL,[BX]
CBW
POP BX
MOV [BX],AL
```

```
OR AX,AX
JNE $+5
JMP _3
JMP _2
_3:
MOV AX,-2[BP]
MOV SP,BP
POP BP
RET
CODE ENDS
END
```

Back to Article

Listing Four

```
SEG CODE include "runtime.i"

SEG XDATA AT MNAME_var_at
SEG CODE

PUBLIC _strcpy:

;;ENTER
POP B
POP ACC
LCALL __enter

;;ADDSP(-2)

MOV DPTR,# 2
LCALL __addsp

;;GETw1s(6)
MOV DPTR,# -3
LCALL __getw1s
;;POINT2s(-2)
MOV DPTR,# 5
LCALL __point2s
;;PUTwp1
MOV DPL,R5
MOV DPH,R4
MOV A,R2
MOVX @DPTR,A
INC DPTR
MOV A,R3
MOVX @DPTR,A

_2:
;;POINT2s(6)
MOV DPTR,# -3
LCALL __point2s
;;GETw1p(0)
MOV DPTR,# 0
LCALL __getw1p
;;INCwp
MOV DPL,R5
MOV DPH,R4
```

```
INC DPTR
MOVX A,@DPTR
ADD A,#1
MOVX @DPTR,A
MOV DPL,R5
MOV DPH,R4
MOVX A,@DPTR
ADDC A,#0
MOVX @DPTR,A
;;PUSH1
LCALL __push1
;;POINT2s(4)
MOV DPTR,# -1
LCALL __point2s
;;GETw1p(0)
MOV DPTR,# 0
LCALL __getw1p
;;INCwp
MOV DPL,R5
MOV DPH,R4
INC DPTR
MOVX A,@DPTR
ADD A,#1
MOVX @DPTR,A
MOV DPL,R5
MOV DPH,R4
MOVX A,@DPTR
ADDC A,#0
MOVX @DPTR,A
;;MOVE21

MOV A,R3
MOV R5,A
MOV A,R2
MOV R4,A
;;GETb1p(0)
MOV DPTR,# 0
LCALL __getb1p
;;POP2
LCALL __pop2
;;PUTbp1
MOV DPL,R5
MOV DPH,R4
MOV A,R3
MOVX @DPTR,A
;;NE10f
MOV A,R3
ORL A,R2
JNZ $+5
LJMP _3
;;JMPm(_2)
LJMP _2

_3:
;;GETw1s(-2)
MOV DPTR,# 5
LCALL __getw1s
;;RETURN(2)
```

```
MOV DPTR,# -2-4
LJMP __return

END
```

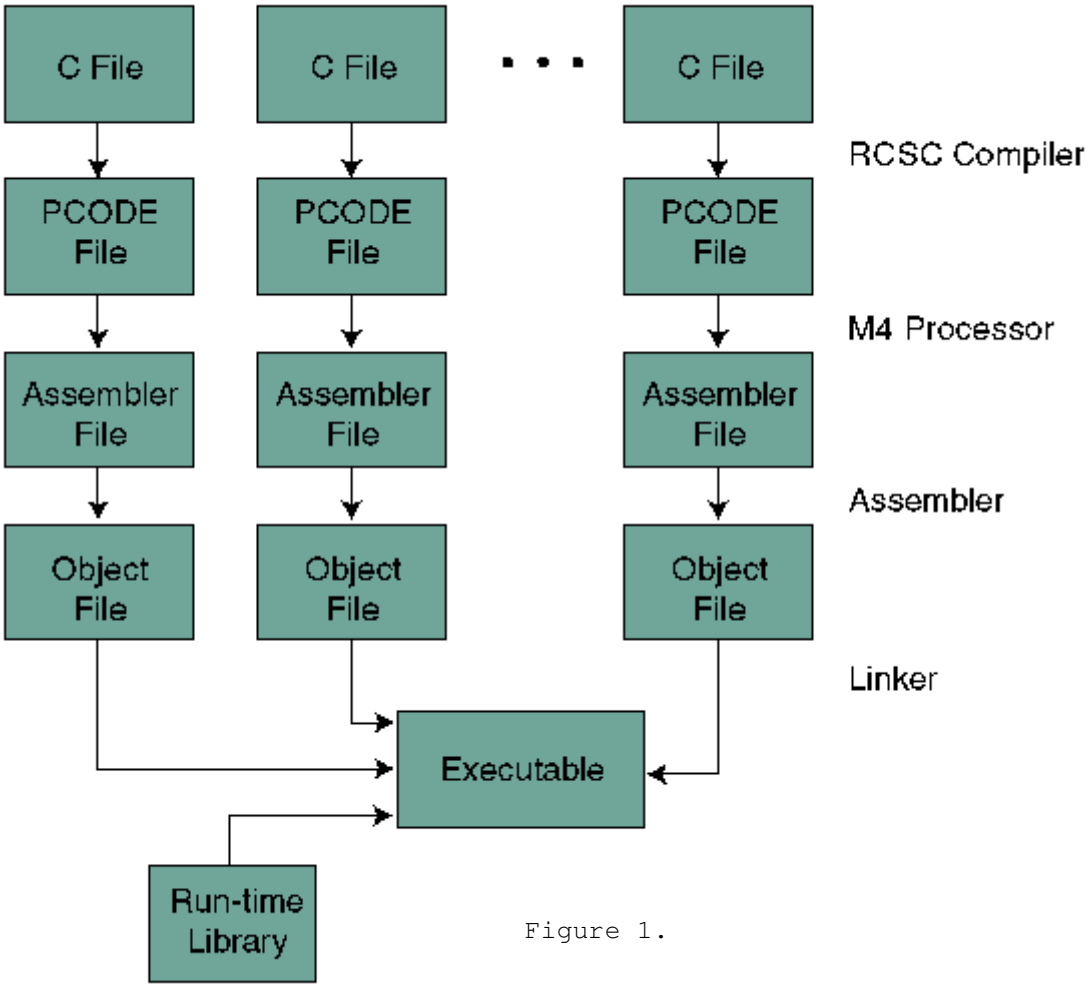

Figure 1.

Table 1.

Pseudo PCODE	Description
BEGINDUMP	Start of literals for preprocessor.
BEGININTR	Start of interrupt information.
BEGINLIT	Start of literals.
BEGINTASK	Start of task declarations.
CSCEXTRN	Includes RCSC external declarations.
CSCTERM	End of a RCSC module.
DECLARE	Declares local label.
DECPUBLIC	Declares global label.
ENDINTR	End of interrupt information.
ENDLIT	End of literals.
ENDSEG	Terminates a segment.
ENDTASK	End of task declarations.
EXTERNAL	Creates external symbol.
INTRS	Defines interrupt information.
MODULENAME	Defines name of RCSC module.
SHADOW	Reserves space for user variables in the XDATA segment.
TOSEG	Changes to a different segment.

Register	Usage
(a)	
AX	Primary register.
BX	Secondary register.
CL	Number of arguments in a function call.
SP	Stack pointer.
BP	Stack frame pointer.
Others	Temporary registers.
(b)	
R0 (bank 2)	Temporary register.
R1 (bank 2)	Temporary register.
R2 (bank 2)	Primary register (high byte).
R3 (bank 2)	Primary register (low byte).
R4 (bank 2)	Secondary register (high byte).
R5 (bank 2)	Secondary register (low byte).
R6 (bank 2)	Temporary register.
R7 (bank 2)	Number of arguments in a function call.
24	Virtual stack pointer (high byte).
25	Virtual stack pointer (low byte).
26	Virtual base pointer (high byte).
27	Virtual base pointer (low byte).
28–31	Multiplication and division registers.
32	Sign of operands and result (signed arithmetic).
33	Port 2 value (during task switching).
34	Accumulator A (during task switching).
35	Accumulator B (during task switching).
36	DPH (during task switching).
37	DPL (during task switching).
DPTR	External memory data pointer.
SP	Hardware stack pointer.

Table 2.

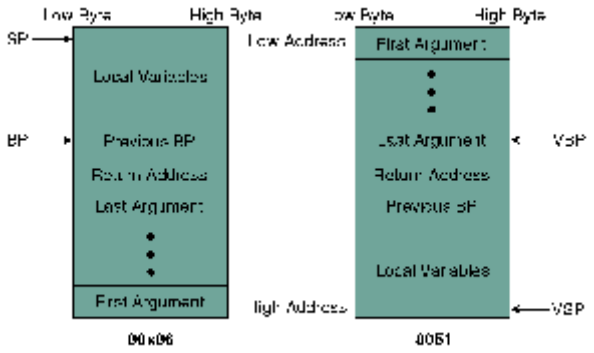

Figure 2.

```
 ...
/* call function */
function(a, b);
 ...
```

(a)

```
function(a, b)
int a, b;
{
int x, y;

#asm
     MOV AX,4[BP] ;move b to AX
     ...
     MOV AX,6[BP] ;move a to AX
     ...
     MOV AX,-4[BP]          ;move y to AX
     ...
     MOV AX,-2[BP]          ;move x to AX
#endasm
}
```

(b)

```
function(a, b)
int a, b;
{
int x, y;
/* use PCODE to access arguments and local variables using
     80x86 offsets. The true offsets are calculated by the
     8051.m4 macro ADJUST automatically */
#asm
     GETw1s(4)              ;;move b to primary register
     ...
     GETw1s(6)              ;;move a to primary register
     ...
     GETw1s(-4)             ;;move y to primary register
     ...
     GETw1s(-2)             ;;move x to primary register
#endasm
}
```

Figure 3.

Table 3.

Interrupt Number	Description
0	External interrupt 0.
1	Timer 0 overflow interrupt.
2	External interrupt 1.
3	Timer 1 overflow interrupt.
4	Serial port interrupt.
5	Timer 2 overflow interrupt.
6	Analog-to-digit converter interrupt.
7	External interrupt 2.
8	External interrupt 3.
9	External interrupt 4.
10	External interrupt 5.
11	External interrupt 6.

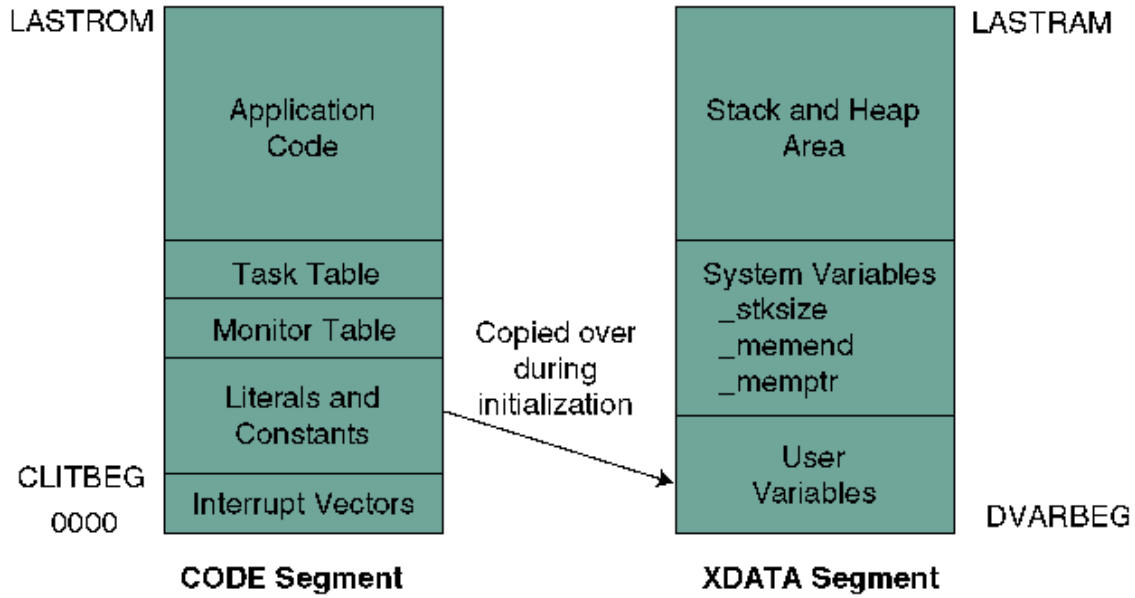

Figure 4.

```
(a)
macroname(arg1, arg2,..., argn)

(b)
define(`OPxy',
`MOV A,$3
ifelse(len($5), 0, `$1 A', `$1 A,$5')
MOV $3,A
MOV A,$4
ifelse(len($6), 0, `$1 A', `$2 A,$6')
MOV $4,A')
```

Example 1.

```
(a)
OPxy(ADD, ADDC, R3, R2, R5, R4)

(b)
MOV  A,R3
ADD  A,R5
MOV  R3,A
MOV  A,R2
ADDC A,R4
MOV  R2,A
```

Example 2.

```
__intr_1:
LCALL   __handler       ;;common interrupt handler
DW      __dodelta       ;;user interrupt function
DW      1                ;;interrupt number
```

Example 3.

(a)

```
vargfunc(argc) int argc; {
  int *nxtarg;
  int *ptr, *ctl;

  /* get pointer to first argument
     */
  nxtarg = CCARGC() + &argc - 1;
  ...
  /* get first argument */
  ptr = *nxtarg--;
  ...
  /* get next argument */
  ctl = *nxtarg--;
  ...
}
```

(b)

```
vargfunc(argc) int argc; {
  int *nxtarg;
  int *ptr, *ctl;

  /* get pointer to first argument
     */
  nxtarg = &argc - CCARGC() + 1;
  ...
  /* get first argument */
  ptr = *nxtarg++;
  ...
  /* get next argument */
  ctl = *nxtarg++;
  ...
}
```

Example 4.

```
vargfunc(argc) int argc; {
  va_list nxtarg;
  int *ptr, *ctl;

  /* get pointer to first argument
     */
  va_start(nxtarg, argc);
  ...
  /* get first argument */
    ptr = va_arg(nxtarg, int);    ...
  /* get next argument */
  ctl = va_arg(nxtarg, int);
  ...
```

Example 5.