Small-C

A Compiler for ORCA/M

by Mike Westerfield

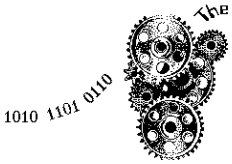


Table of Contents

Chapter 1 - Introducing Small-C

Who Should Use Small-C

Small-C is basically an integer-only subset of the popular C programming language. As such, it shares a number of the strengths and weaknesses of C. C is generally used as a systems programming language, for those applications where a high level language must be used, yet efficiency is still of the utmost importance.

C shares many structural similarities with modern block-structured languages like Pascal, yet there are a number of features from Fortran. Unlike Pascal, C makes no attempt to protect the programmer from his mistakes - and this is perhaps the most distinctive feature of the language. C assumes the programmer knows what he is doing, and lets him get away with murder You can write self modifying code with C, index outside of array bounds, access a real number as an integer, and in general do things which, if you are not careful, will give very unpredictable results. The other side of all that is that you can do some very clever things with C, cutting the size and execution times of a program significantly. C, then, is a language for experienced programmers who are willing to sacrifice some safety for the very best performance a compiler can give.

Small-C is a subset of the full language. Specific differences will be discussed later. For someone writing production code, Small-C will be a risky choice. Small-C is primarily intended to fill three needs:

1. Small-C shows how to install a compiler in the ORCA/M environment, and thus serves as an example to compiler writers who would like to bring a compiler up on ORCA/M.

2. Small-C comes with source code, and is written in itself. This makes it a valuable tool for anyone who would like to learn more about compilers, or who needs to implement a special purpose language.

3. Small-C, while not a full implementation, will be useful to those needing a compiler until full languages are available on ORCA/M.

If you fall into one of those categories, then Small-C is for you. If you do not, another compiler would probably be a better choice.

Small-C and Full C

This section is intended for those who already know C. If you do not, skip it for now. After learning Small-C, you can come back and reread this section to find out what the differences are between Small-C and full C.

As was mentioned before, Small-C is basically an integer-only subset of C. Small-C only includes three data types: the integer, the character, and pointers to either of these. Pointers use two bytes of memory. Characters use one byte of memory. Two and four byte integers are available, although all math is done using four byte mathematics routines.

The array is the only structured data type available. All arrays are singly subscripted. Arrays must be arrays of integers or characters - arrays of pointers are not allowed.

Small-C allows functions, but does not recognize typed functions. In Small-C, all functions return integers. Passed parameters are allowed, and follow normal rules.

Like full C, global variables can be defined anywhere in the program, so long as they are not inside of a function definition. Global variables are assigned fixed memory locations. They cannot be explicitly initialized, but due to the fact that they are allocated space via the ORCA/M DS directive, all start out as zero.

Functions are fully recursive, since local variables and passed parameters are allocated on a stack frame. Local variables are not initialized in any way. All local variables must be defined before any statements in a function - it is not possible to define local variables within a compound statement.

All of the statements from full C are available in Small-C.

Almost all of the operators from full C are in small C. The direct selection operator (.) and the indirect selection operator (->) are missing, but have no use without structures, anyway. Casts are also missing, but again, have little use when only characters and integers are available. The sequential evaluation operator (,), the conditional operator (? :) and the assignment operators that combine an operation with the assignment statement (+= - *= /= %= <<= >>= &= |=) are the only missing operators that might have some use in Small-C.

In the preprocessor, there are only two commands from full C, and both have additional restrictions imposed. The #include facility is restricted to using quotes around the file name, so-there is no default library directory. In addition, an included file is not allowed to use the #include directive to include another file. #define is available, but it cannot use parameters, so it is restricted to simple text substitutions.

Small-C has a few minor additions to the full language, all of which can be avoided if portability is an issue. The first is the #asm-#endasm directives, allowing inline assembly language. The second is a list of directives to make interfacing with the ORCA environment a little easier.

Chapter 2 - Setting Up Small-C

Small-C is designed to fit right in to the ORCA language environment. It comes on two disks. The first disk, Small-C, contains the files needed to compile Small-C programs. The second disk, which contains source code, does not concern us for now. It is discussed in Chapter 5.

The description that follows assumes that you have already initialized ORCA/M, and are reasonably familiar with that program.

The first step is to copy the compiler to the language prefix. The language prefix is where you have the assembler and linker (ASM6502 and LINKER). On the original distribution disk, this was /ORCA/LANGUAGES. The compiler is called CC. For example, if you are using a copy of the original distribution disks for ORCA/M, type

```
COPY /SMALL.C/CC /ORCA/LANGUAGES
```

The next step is to install the command CC as a language name. This is done with the COMMANDS utility. After typing

```
COMMANDS
```

(and possibly putting the /UTILITIES disk online) you will see a list of the current command table. Create a new command called CC, and identify it as a language with a language number of 7. Exit the COMMANDS utility and execute SYSGEN. make some change - it really doesn't matter what - then change things back the way they were and exit SYSGEN, saving the new system back to your boot prefix. Refer to the ORCA/M reference manual if you do not know how to use COMMANDS and SYSGEN.

The third step is to move the subroutine libraries into place. The subroutine libraries are in a file. called A..SMALLC.A. Copy this file to the subroutine library prefix (/LIBRARY on the ORCA distribution disks) and use COMPRESS to alphabetize the libraries. This is an important step - if the libraries are not in alphabetical order, you will get link edit errors.

There are three macro libraries * one of these should be on the current prefix when doing a compile. C.MACROS will cause Small-C to produce native code programs, which are much faster than the alternative. For that reason, you should probably start with C.MACROS. C.PCODE causes

Small-C to generate P-code, and link in an interpreter. For large programs, P-code will be much smaller than native code. To generate P-code programs, copy C.PCODE to the prefix that you will compile from and rename it to be C.MACROS. The last macro file is called C.ABSOLUTE. It also generates P-code programs, but is tailored for compiling large programs from floppy disks. It is discussed in Chapter 5. The subject of P-code vs. native code is discussed more fully in Chapter 5.

with these steps completed, you are ready to start writing C programs. The next chapter is a brief tutorial introduction to Small-C. If you are new to C or ORCA, start there. If you are already familiar with both C and ORCA, you may want to skip to Chapter 4, which is a reference manual for the Small-C language.

Chapter 3 - Small-C Tutorial

The First Program

In this chapter, we will learn Small-C using a series of increasingly sophisticated examples. The pace will be fast - it is assumed that you already know assembly language, and are therefore a fairly advanced programmer. Those wishing a more gradual pace can consult any of the fine books now on the market that teach C programming.

We'll start right in with a Small-C program, and look at what we did after it runs. The first step is to make sure that all of the initialization from Chapter 2 has been performed. In particular, C.MACROS should be on the current prefix - i.e., you should see it when you catalog the disk.

The first step will be to enter the program. Each file created by the editor is stamped with a language ID number. This number tells the operating system which compiler or assembler to use to compile the file. If you do nothing, the Small-C program will be stamped as an ASM6502 file - and the assembler will have no idea what to do with the Small-C program. The first step, then, is to type CC. This tells the system that the next file created with the NEW command is a C program. Now enter the editor in the normal way and enter the following program exactly as it reads. Be sure and use lowercase letters!

```
#keep "tryit"
main ()
{print("Hello, world ... \r");}

print(str)
char *str;
{while(*str)putchar(*str++);}
```

Exit the editor and save the file using the file name PROG1. Then type RUN PROG1, just like you would for an assembly language program. Small-C will compile the program, producing an assembly language file called TRYIT. The assembler will be called automatically to assemble the program, after which the link editor will create an executable file, also called TRYIT (deleting the intermediate code file, but it was no longer needed).

Finally, the program will be executed, printing

```
Hello, world ...
```

on the screen. If this did not happen, Small-C is not initialized properly. Review each of the steps from Chapter 2 and this section and make sure they are all
done correctly,

Functions

it w ill be a while before we understand all of the features of that first program. The second routine, though, is a good one to keep around - it lets you write any string to the CRT. Incidentally, routines are called functions in C. This is because every subroutine returns a value - even if we do not ask it to. More on that later.

For right now, it is important to learn the structure of C programs. Like assembly language with ORCA, C programs are made up of a series of program segments. Each has a name, which appears as the first token of the function. Right after the name is a parameter list. The parameter list is a series of names enclosed in parenthesis. Look at the first function, `main`. Note that it has a set of parenthesis, even though there are no parameters. C requires the parenthesis, whether or not there are any parameters.

After the function header, any parameters used in the header must be defined. In Small-C, the parameters must be defined in exactly the same order that they appear in the parameter list. The compiler does not check your work here, so be very careful! Finally, the function body appears. The function body starts with a t, and is followed by local variable definitions and a series of statements. The whole thing finishes up with a }.

You may notice in the example above that the first function is called `main`. Although things would have worked just fine if `main` were the second function, it is critical that there be a function somewhere in the program called `main`. The name must be spelled with lowercase letters. That function is the first one executed when your program starts.

Global Variables

our second program will introduce the concept of global variables. It will be fairly simple, and a little contrived. We will set a global variable to a value, then call a second function that will print that value by calling yet a third

function. The third function is actually useful. It is called printint, and we will use it to print integers.

```
#keep "tryit"
int i;
main()
{i = 0;
while (i < 20)
    {doit();
    i++;
    }
}

doit ()
{printint(i);
putchar('\r');
}

printint(i)
long i;
{long den;

if(i)
    {if( i < 0 )
        {putchar('-');
        i = -1;
        }
    den 1000000000;
while(i < den) den = den/10;
while(den)
    {putchar(i/den+'O');
    i = i%den;
    den = den/10;
    }
else putchar('O');
}
```

Type it in and run the program. There are several things to learn from this program. First, global variables are any variable defined outside of a function. These variables are available to any function. What about `printint`? It uses `i` as a parameter. The answer is that local variables have precedence if they have the same name as a global variable. Note that a global variable does not have to be defined before the first function - it can

be defined between two functions as well. The only restriction is that it must be defined before the first function that will use it. Unless there is good reason to do otherwise, it is probably a good idea to put global variable definitions at the start the program - you know, style.

The global variable shown is an integer. You can tell this because the line that defined it starts with the reserved word `int`. You can also define character variables by changing the keyword to char. Instead of `int`, you could also use `short`. In Small-C, short integers and the default integer size are both two bytes long. To get a four byte integer, `long` is used. The printint function defines `den` and `i` as long integers. More than one variable can be defined on a single line, too. To do that, just separate the variables by commas. The entire definition must be followed by a semicolon (as all C statements are).

Incidentally, you don't need to worry about mixing data types. Small-C automatically converts two byte integers, characters and pointers to four byte integers before using the values.

Local Variables and Parameters

Local variable definitions look just like global variable definitions. The only difference is that a local variable is defined inside the function that it is local to. Only the function that defined the variable can use it. There are actually two kinds of local variables, since parameters are also local to the function that defines them.

Passed parameters in C are always passed by value. That means that a function cannot change the value that is passed to it. (Later, we will see a way to get around this.) The parameter name must appear in two places. In the parameter list, the name of the parameter appears, with parameters listed in the order in which they are specified when the function is called. After the parameter list and before the character that starts the function body, each of the parameters must be defined. The parameters must be defined in the same order that they appeared in the parameter list! As seen in our examples, the actual definitions look just like global definitions.

Local variable definitions come right after the opening [, but before any statements. Our examples show that these look just like global variable definitions as well. The only difference is that only the function where they are defined can use them.

Recursion

C is a recursive language. When a functions is called, it sets up a special area for all of the passed parameters and local variables. If the function calls itself, a second area is set up, and each of the parameters and local variables are redefined.

A few sections ago, we said that all code segments are functions in C, and that they return a value. That means that a functions can be used in an expression. The return statement lets you return a value. The value to return appears right after the keyword `return`. It can be left out - in that case, you still return from the function, but its value is not predictable. In either case, the return statement ends with a semicolon. Also note that C is case sensitive, and that `return` is written in lowercase! `RETURN` or `Return` will not work, only `return`.

The following example combines the new return statement with recursion to print the factorials that can be represented by the four byte integer size supported by Small-C. The function `printint` from our last example is used to print the values.

```
#keep "tryit"
main()
{long i;

for  (i = 1; i <= 12; ++i)
    {printint(i);
    putchar(' ');
    printint(factorial(i));
    putchar ('\r');
    }
}

factorial(i)
long i;
{
if (i == 1) return 1;
else return factorial(i-1)*i;
}
```

```
printint(i)
int i;
{long den;

if(i)
    {if(i < 0)
        {putchar('-');
        i = -1;
        }
    den 1000000000;
    while(i < den) den = den/10;
    while(den)
        {putchar(i/den+'O');
        i = i%den;
        den = den/10;
        }
    }
else putchar('O');
}
```

The if Statement

We have already used the if statement in some of the program examples - now we will take the time to examine it in detail. Like most high level languages, the if statement is used to conditionally execute a statement. The `if` keyword is followed by an expression, enclosed in parenthesis, and a statement. The expression is evaluated. If the value of the expression is not zero, the statement is executed. if the value of the expression is zero, the statement is not executed.

There are several unique and important points here. First, all expressions used to evaluate a condition in C are enclosed in parenthesis. Since the closing parenthesis marks the end of the condition, there is no need for a then keyword. Second, C does not have a Boolean variable. In fact, there is very little distinction between characters and integers. If a value is zero, C treats it as being the same thing as false. If it is not zero, it is treated as being true. Also, an assignment statement is not a statement at all, but an expression, so you can assign a value inside of a condition.

Here are the conditional operators in C, as well as what they test for:

Operator	Condition
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
==	equal
!=	not equal

As a test of what was just said, consider the value of `var` after the following statement is executed:

```
if (var = 4) var = var+1;
```

The correct answer is 5. When the condition is evaluated, `var` is assigned the value 4. This is also the result of the expression, and since it is not zero, it is treated as being true, so one is added to `var`. To get the effect that was probably intended, you would write

```
if (var == 4) var = var+1;
```

Like Pascal, C if statements can be followed by an else clause. If the expression evaluates to zero, the statement after the else is executed. For example, using our print routine, the following function will write out a Boolean value.

```
printbool(b)
int b;
{if(b) print("true");
else print("false");
}
```

If you are a Pascal programmer, there are two stumbling blocks here that a BASIC programmer may never have a problem with. The first is that strings in C are surrounded by the " character, not the ' character. The ' character is reserved for character values, not string values. Character values are converted to their integer equivalent and act like integers in expressions. Strings cannot be used in an expression at all. The other point is that each C statement ends with a semicolon, and that semicolon is required. In Pascal, the semicolon is not a statement terminator, it is a statement separator. In the function above, leaving out any one of the semicolons would give an error.

Looping Statements

There are three kinds of looping statements in C. The easiest is the while loop, which starts with the keyword while, is followed by a condition, and ends with a statement. Like the if statement, the condition is an expression enclosed in parenthesis. The condition is evaluated. If it is true (non-zero), the statement is executed and the process repeats. if it is false, the statement after the while loop is executed. For example, the following sequence will use our `printint` function to count to 100.

```
main()
{int i;

i = 1;
while (i <= 100)
    {printint(i);
    putchar('\r');
    i++;
    }
}
```

How does the while loop ever finish? After all, it is the statement after the while loop that gets executed. The answer lies in the brackets that surround the three statements. This causes them to be considered as a block of statements, like `begin-end` in Pascal. The effect is that the call to `printint`, the call to `putchar` and the assignment to `i` are executed before the condition is retested. This is called a compound statement. A compound statement can be used anywhere that a simple statement can be used.

The next looping statement is a lot like the first. The difference is that the condition is tested after the statement has been executed once, not before. It is called the `do-while` loop. Pascal programmers should note that the loop is re-executed if the condition is true, not if it is false. Our last example, written with the `do-while` loop, would be

```
main()
{int i;

i = 1;
do
    {printint(i);
    putchar('\r');
    i++;
    }
while (i <= 100);
}
```

The only tricky thing to note is that the condition must be followed by a semicolon, due to the fact that there is no statement there.

The last looping statement is the one that will probably be familiar in some form to everyone who has used a high level language. On the other hand, it is very different from what you may be used to. The for statement is used for repetitive looping in C. Our example, rewritten with a for statement, is

```
main()
{int i;

for  (i = 1; i <= 100; i++)
    {printint(i);
    putchar('/r');
    }
}
```

The syntax is a bit odd, so let's look at it carefully. Inside the parenthesis are three expressions separated by semicolons. After the expressions is the statement that will be executed while the loop is looping. The first expression is executed once, as the loop starts. It is used to set up the loop. The next expression is a condition that determines when the loop will stop. The condition is tested before the statement is executed, so it is possible to have a for loop that does not execute the statement at all. After testing the termination condition, the statement is executed, and finally the last expression is evaluated. The last expression is used to update the loop counter.

The uses for each of the expressions are simply accepted standard uses. There is really nothing to keep you from putting an expression in that has

nothing to do with the for loop. For example, this function will do the same thing as the one shown above:

```
main()
{int i;

for (i = 0; i < 100; printint(i))
    {putchar('\r');
    i++;
    }
}
```

In fact, the for loop in C is defined in terms of the while loop. The first for loop is exactly equivalent to our example at the start of the section that showed a while loop!

The goto Statement

C supports the goto statement found in most high level languages. It consists of the keyword `qoto`, followed by a label and a semicolon. The next statement executed is the one after the label that is branched to. Labels in C look like identifiers - they must start with a character and be followed by alpha-numeric characters. Numeric labels are not allowed.

To define the label, simply code it followed by a colon. The label must be followed by a statement! if you are branching to the end of a subroutine, as in the example below, use the null statement - which is simply a semicolon.

Note that Small-C requires that the colon follow right after the label, with no intervening spaces.

```
main()
{int i;

i = 0;
start: if (i >= 100) goto end;
i++;
printint(i);
putchar('\r');
goto start;
end:;
}
```

The switch Statement

The switch statement is C's answer to the case statement of Pascal or the computed goto of BASIC and FORTRAN. It actually works more like a computed goto than a case statement, but looks more like a case statement than a computed goto!

Let's start by examining how a switch statement is with the keyword `switch`, followed by an expression enclosed in parenthesis. This is followed by a single statement, which is almost always a compound statement.

When the switch statement is encountered, the expression is evaluated. C then branches to the case label that matches the value of the expression. A case label is the keyword `case`, followed by a constant and a colon. Case labels are only allowed in the statement after a switch statement.

If there is no matching case label, the statement after the expression is skipped, unless a default label is coded. A default label takes the form of the reserved word `default` followed by a colon. only one default label can be used in a given switch statement.

Note that, in both the case label and the default label, Small-C requires that the colon follow the label immediately, with no intervening spaces.

A very important point is that the switch statement simply does a goto when branching to the case label or default label. Try the example below, and note that the input number determines how many asterisks would be printed. The point is that there is no imbedded goto before a case label - execution simply falls through the remainder of the statement!

```
main()
(int i;
for (i = 1 ; i < 6; i = i+1}
    {switch (I)
        {default:
        case 4: putchar('*');
        case 3: putchar('*');
        case 2: putchar('*');
        case 1: putchar('*');
        }
    putchar('\r');
    }
    }
```

break and continue

The situation with the switch statement would be a bit discouraging were it not for two other C statements that save us from the goto. The break statement is simply the keyword `break` followed by a semicolon. When used, it exits the while, do-while, for loop or the switch statement that it appears in. The continue statement is the keyword `continue` followed by a semicolon. It can only be used in the looping statements (while, do-while and for). It causes execution to continue from the end of the loop.

For example, we could rewrite the switch example as

```
main()
{int i;

for (i = 1; i < 6; i = i+1)
    switch (i)
        {default:
        case 4:  print("****\r");
                 break;
        case 3:  print("***\r");
                 break;
        case 2:  print("**\r");
                 break;
        case 1:  print("*\r");
        }
}
```

Arrays

Small C supports singly subscripted arrays. An array is defined by placing a pair of square brackets around the number of array elements needed, and putting this after the name of the array when it is defined. Arrays can be global, local, or passed as parameters. Note that when an array is passed as a parameter, it is always passed by reference, not by value - this means that passed arrays are not recursive, and that if a function changes an array value, it is changed for the calling function as well! our example illustrates that principal.

```
#keep "tryit"
main()
{int a[10];

init(a);
printarray(a);
}

init (arr)
int arr[];
{int i;

for (i = 0; i < 10; ++i) arr[i] = i+'O';
}

printarray(arr)
int arr[];
{int i;

for (i = 0; i < 10; ++i)
    {putchar(arr([i]); putchar('\r');}
}
```

Notice that the array was defined as a[10]. Later, we access the element a[0]. That is fine, since all C arrays are indexed from 0. If you recall, though, we said that the value between the square brackets is the number of array elements. Note that it is not the highest legal array subscript! Trying to access a[10] will cause a great deal of trouble. a[9] is the highest legal subscript in our example.

Also, notice that when an array is passed as a parameter, the integer between the square brackets is optional. C does not prohibit indexing outside of the legal bounds of an array, so even if a subscript is coded, it is ignored. The brackets let the function know that the parameter is an array; you can use whatever subscript you like.

Pointers

Perhaps the greatest strength of C is its remarkable pointer handling capabilities. Small-C supports pointers to characters or integers. Like full C, the name of an array is also a pointer - it is basically a read only pointer pointing to the first element in the array.

A pointer always requires two bytes of storage. When it is defined, it appears as an asterisk followed by the name of the pointer. If we have defined

```
int *ptr;
```

then `ptr` refers to the current value of the pointer, and `*ptr` refers to the integer value that the pointer points to.

It is legal to add, subtract, increment and decrement pointers. When a pointer is a pointer to a character (which requires one byte of storage), these operations are identical to the same operations on an integer. When the pointer is a pointer to an integer (which requires two bytes of storage), the amount to change the pointer by is first multiplied by two. For long integers, the pointer would be changed by four, rather than two. Thus, adding one to a pointer will cause it to point to the next data element of the kind that it points to.

This makes pointers work a great deal like arrays, and C in fact allows array and pointer accessing in much the same way. In C, if `ptr` is a pointer, or the name of an array, then `ptr+4` is completely equivalent to `ptr[4]`.

An additional operator which is a great help is the & operator. When any identifier is preceded by a &, the value is the address of the location where the value is stored, not the value itself. For example, assuming `ptr` is still a pointer to an integer, the following is equivalent to `i = i*2`.

```
ptr = &i;
ptr[01 = *ptr*2;
```

Expressions

Up to now, we have pretty much ignored the exact representation of an expression in C. This is because they work pretty much like expressions in other high level languages. Other than the pointer operators & and *, and the fact that = is an operator in an expression, things work very much like BASIC, Pascal and FORTRAN.

The use of the = operator has been alluded to , but never fully explained. In C, the = operator requires an l-value on the left, and an expression on the right. An l-value is basically the C term for something that you can assign a value to. For example, 4 is not an l-value - you cannot change a constant. A variable, pointer, or array element, however, is an l-value, since values can be assigned to all of these.

Note that, since the = operator is an operator, a = 4 is an expression. Also note that the = operator is followed by another expression. That means that

```
a = b = c = d = e = 0;
```

is perfectly legal in C - all of the variables are set to zero. It is also more efficient than five separate assignment statements.

All expressions in C return a value. Thinking about how the above example must work, it is clear that the value of an expression containing an assignment operator is the value of the expression to the right of the = character.

Other than those covered so far, the only really new operators in C are the increment and decrement operators. If an l-value is preceded by a ++, it is incremented, and the incremented value is returned. Following an l-value by a ++ still increments the l-value, but the value used in the expression is the original value. For example:

```
/* sample 1: */
i = 5;
a  = i++;

/* sample 2: */
i = 5;
a = ++i;
```

In both samples, `i` ends up with the value 6. In the second sample, so does `a` - but the first case will not set `a` to 5, since the increment occurs after `i` is used the expression.

The `--` decrement operator works like the increment operator, except that the value is decremented.

Incrementing or decrementing a pointer to an integer will change its value by two, so that the pointer points to the next (or last) integer, not the next byte. Incrementing or decrementing a pointer to a long integer changes the pointer by four.

Below are all of the operators available in C. The precedence tells which operator is evaluated first - precedence is what cause `1+2*3` to be 7, instead of 9. Parenthesis can be used in the normal way to override operator precedence, so `(1+2)*3` is indeed 9.

Operator	Precedence	Description
-	1	unary minus
!	1	logical negation
~	1	bitwise negation
*	2	integer multiplication
/	2	integer division
%	2	integer modulus
+	3	integer addition
-	3	integer subtraction
<<	4	bit shift left ("multiplies" by powers of 2)
>>	4	bit shift right ("divides" by powers of 2)
<	5	test for less than
>	5	test for greater than
<=	5	test for less than or equal
>=	5	test for greater than or equal
==	6	test for equality
!=	6	test for not equal
&	7	bitwise and
^	8	bitwise exclusive or
|	9	bitwise or
&&	10	logical and
||	11	logical or
=	12	assignment

Table of C Operators

Interfacing to Assembly Language

Interfacing to assembly language with Small-C is actually quite simple. To call an assembly language function from Small-C, simply code a function call just as if you were calling a Small-C function, but don't include the function in the C part of the program. The compiler assumes that the linker will find the function. (Note: this is different from full C, where the `extern` qualifier is needed to prevent the compiler from flagging an error.)

The assembly language part of your program can be appended right onto the end of the C part. The last include directive in the C program should point to the. assembly language file. The compiler will recognize that the file is

not a C file due to the language identification number, and pass control on to the assembler.

The subroutines that are actually being called should start with NOP instructions. The NOP happens to correspond to the P-code that means "switch from executing P-code to executing native code." A jump to SYSCRET returns control to the C function that called you. A subroutine written this way will work with either native code or p-code.

Global variables defined within the C program can be accessed from any assembly language subroutine that has a USING SYSCCOM directive. The assembly language subroutine can read or write any of these values.

Accessing passed parameters is a little more difficult. At $98 is a pointer which points to the top of the evaluation stack. The pointer points to the last used byte. Each parameter requires four bytes of storage. They are stacked in the same order that they were coded. Integers, characters and pointers are passed by value - the number on the stack is the actual value of the integer, character or pointer. Arrays and strings are passed by reference - the address of the first element of the array or of the first character of the string is passed. (Strings appear in memory as a series of ASCII characters followed by a $00 byte.) All stacked values are four byte integers. The stack itself starts at a high memory value and builds down, much like the hardware stack on the 6502.

Finally, the assembly language routine may want to return a value. To do this, simply place the return value on top of the stack before the jump to SYSCRET. Remember - the value must be a four byte value. For example, the following assembly language subroutine implements a search for a character in a string. A call from C might look like

```
location = search ("This is a test string",'s',14);
```

The search starts at the value specified by the integer, and the string is searched for the character. If it is found, the location is returned, otherwise a zero is returned. The assembly language function is:

```
************************************************
*
*  SEARCH - Search a String for a Character
*
*  Inputs:
*        TOS-2 - address of string
*        TOS-1 - character to search for
*        TOS - start location
*
*  Outputs:
*        SEARCH - location of character
*
*  Notes:
*        Limited to 255 character strings
*
************************************************
*
SEARCH    START
STP       EQU   $98          STACK POINTER
DISP      EQU   $02          ADDR OF STRING
CH        EQU   $06          CHARACTER TO SEARCH FOR
STAD      EQU   $OA          START DISP FOR SEARCH

          NOP                SWITCH TO ASSEMBLY
          MOVE  (STP),DISP,#12 RECOVER PASSED PARMS
          LDY   #0           SKIP TO START POINTr
          LDX   DISP          QUITTING IF IT IS PAST
          BEQ   LB2           THE END OF THE STRING
LB1       LDA   (STAD),Y
          BEQ   NOPE
          INY
          DEX
          BNE   LB1
LB2       LDA   (STAD),Y     QUIT IF AT END OF STRING
          BEQ   NOPE
          CMP   CH           CHECK FOR MATCH
          BEQ   MATCH
          INY              CHECK NEXT CHAR
          BNE   LB2
NOPE      LDY   #$FF       MATCH NOT FOUND ->
!                             RETURN 0
MATCH     INY              MATCH FOUND ->
          TYA                RETURN LOCATION
```

```
          LDY   #0
          STA   (STP),Y
          TYA
          INY
          STA   (STP),Y
          INY                 UPDATE STACK POINTER
          STA   (STP),Y
          INY
          STA   (STP),Y
          JMP   SYSCRET       RETURN TO C PROGRAM
          END
```

In addition to calling assembly language subroutines from Small-C, it is also possible to insert assembly language statements in line in a C function. To do this, code the `#asm` directive. This is followed by the assembly language statements. To switch back to

C, code the #endasm directive. Generally, this should only be done if native code is being used. If p-code macros are being used, it will be necessary to write the assembly language portion using the intermediate code instruction set, described in chapter 5.

As an example, the following functions sets up high resolution graphics by clearing the screen and setting the pen position (x and y) to 0.

```
initgraph()
{
x = x = 0;                      /* initialize cursor */
#asm                            /* clear screen */
          LDA   #$20
          STA   1
          LDA   #0
          STA   0
          TAY
LB1       STA   (0),Y
          INC   0
          BNE   LB1
          INC   1
          LDX   1
          CPX   #$40
          BNE   LB1
#endasm
```

Chapter 4 - Small-C Reference Manual

Lexical Issues

Small-C programs, like those of most high level languages, are written as a series of tokens. Each token is a unit of information, analogous to words or punctuation marks in English.

All of the "words" that are predefined are written entirely in lowercase letters. This is a requirement of C - the language is case sensitive, and so does not recognize "A" and "a" as being the same character.

Small-C is fairly lenient about its predefined words. They are not actually reserved, and if you are very careful, they can be used as symbols. Since full C makes these reserved words, it is not a good idea to use them as symbols. Hereafter, they will be referred to as reserved words in Small-C.

The reserved words are:

```
asm       break     case      char      continue
do        else      for       goto      if
int       long      return    short     switch
while
```

In addition, the following are reserved words in full C, and so should not be used in Small-C programs:

```
auto      default   double    enum      extern
float     register  sizeof    static    struct
typedef   union     unsigned  void
```

Note that `asm` is not a reserved word in full C, and that `entry` and `fortran`, while not reserved words in standard C, are reserved in many implementations.

Small-C recognizes the following tokens. Except for comments and string and character literals, these are the only non-alphanumeric characters allowed.

```
!       %       ^       &       *
~       +       =       ~       |
<       >       /       ++      --
<<      >>      <=      >=      ==
!=      &&      ||      (       )
[       ]       ;
```

Small-C recognizes three kinds of constant. The first is the integer, which is a sequence of digits. Since full C treats an integer starting with a zero as an. octal number, it is a good idea not to start integers with a zero unless the value is zero.

Character constants consist of any keyboard character except the single quote mark, surrounded by single quote marks. More than one character can be enclosed in the quote marks, in which case the last is placed in the least significant byte of a four byte integer, the next to last character is placed next to the least significant byte, and so on for the third and fourth characters. The remainder are ignored. Values from the ASCII character set are used. Thus, 'a' is equivalent to the integer 97, and 'ab' is equivalent to 24930 (which is 256*97+98).

String constants consist of a series of characters enclosed in double quote marks. A string is stored in memory as a series of one byte ASCII character code values, terminated by a zero byte. The double quote mark cannot be used inside a string.

In both character constants and strings, the backslash character has special significance. It is used to allow placing characters inside of a string that normally could not be placed there. The character right after the backslash determines what value will be placed in the character constant or string. The characters recognized, and the values they represent, are:

Character	Value	Comment
n	10	newline
t	9	tab
v	11	vertical tab
b	8	back space
r	13	carriage return
f	12	form feed
\	92	character
'	39	character

"	34	character

If the character after the backslash is not one of t above characters, the backslash is treated as a character. The specific effect of following the backslash by a character not in the above table is defined by C, so it is a good idea to always follow backslash by a character from the table.

User defined tokens within C always start with an alphabetic character and are followed by one or more alphanumeric characters. C is a little unusual in that the "alphabetic characters" include the underscore, and in that uppercase and lowercase letters, while both allowed, are distinct. Although symbols can be as long as desired, they must be distinct in the first eight characters.

All of the following are valid C symbols, and all are different.

```
main      MAIN      Main      n14
averylongsymbol     with_underscore
_start
```

Although Small-C considers the case of a letter to be significant, ORCA/M does not. This is important, since Small-C produces assembly language files that are then assembled by ORCA/M. If a symbol is global or if it is the name of a function, it must be resolved by ORCA/M. If that is the case, make sure that the symbol is distinct from all other symbols, regardless of case. For example, do not create a program with two functions, one called `main` and the other called `MAIN`.

C allows comments anywhere that a space would appear. A comment is any sequence of keyboard characters surrounded by a `/*` at the beginning and a `*/` at the end. Comments can extend over more than one line. They are not recursive - that is, `/* /* */ */` is not a legal comment.

The Preprocessor

The Small-C preprocessor is logically separate from the compiler itself. Its functions are performed before the compiler gets a chance to look at the line. This distinction is more important in full C, where a larger variety of preprocessor commands are available, but it is still important to keep this fact in mind when thinking about the `define` directive.

There are a total of six preprocessor commands. These can be grouped into two categories. The first category contains those commands which are found in full C. This includes `define` and `include`. The second category consists of those commands which have been added to make it easier to work in the ORCA programming environment. While these are quite useful, they are not found in C on other operating systems, so they should not be used in code that will be ported to other machines. None are actually required to use Small-C. These include `list`, `keep`, `source`, and `org`.

The preprocessor commands are described in the paragraphs that follow. Each starts with a line that shows how the command would appear in a Small-C program. Although Small-C makes no strict requirement, most full C implementations require a directive to start in column 1 of the source line. Thus, it is a good idea to start commands in column 1. Small-C requires that the name of the command follow the # character with no intervening spaces. Small-C will ignore anything that appears after a valid preprocessor command on the line. Full C will generally give an error if anything but a carriage return appears on the line.

`#define mac string`

This command defines a simple text substitution macro. `mac` should be replaced by the name of the macro, which must be a valid identifier. After skipping any spaces, the string that appears between `mac` and the end of the line will be substituted for `mac` wherever `mac` is used in a program. This is the method used to define constants in C. It can also be used as an effective way to document code.

An example of the define command would be defining the constants true and false:

```
#define true 1
#define false 0
```

`#include "file-name"`

The `include` command causes Small-C to process all of the lines in the included file, then return to the original source file. Included files may not include other files. `File-name` can be any valid ProDOS path name. The quote marks surrounding the file name are required.

If the file to be included is not a Small-C source file, Small-C will terminate all processing and transfer control back to ORCA/HOST. The end of the intermediate file will include an APPEND directive to cause the assembler to process the file. This gives an effective way of combining Small-C and assembly language in a single program. For an example, see the compiler.

`#keep "file-name"`

The intermediate code is sent to the named output file, and when the compile is complete, it is passed on to the assembler. Assuming successful compilation and assembly, the final executable program will have the name given, and the intermediate code file will automatically be deleted.

As with `include`, the quote marks are required. The file name can be any valid ProDOS path name, so long as the file name portion of the path name itself is ten characters or less. For example, /PROFILE/LONGNAME is fine, but /PROFILE/VERYLONGNAME is not. The reason for this restriction is that ORCA/M will need to append ".ROOT" to the file name you supply, and ProDOS allows a maximum of fifteen characters in a file name.

This command must appear before the first function, and cannot appear more than once. It cannot be used if the KEEP parameter is used on the COMPILE command.

`#list on`

The `list` command can take an operand of `on` or `off`. If the operand is `on`, all subsequent source lines are sent to the current output device. If the operand is `off`, subsequent source lines are not listed. The default is on, listing all source lines.

`#org val`

`val` is any constant value. Expressions are not allowed. The finished program will be located at the memory location specified by `val`. This command must appear outside of a function.

#source off

The operand can be on or off, and defaults to `off`. If `on`, all source lines are placed in the intermediate code file as assembly language comment

statements. This is useful for studying the code produced by the compiler, or for hand optimizing the output of the compiler. If it will be necessary to look at the intermediate code file, do not do a link edit - doing a link edit will delete the intermediate code file!

The Program

A C program consists of a sequence of global variable declarations and function declarations. Exactly one of the functions must be named `main`.

Variable Declarations

Variable declarations always start with one of the keywords `int`, `short`, `lonq` or `char`, and are followed by one or more variables, separated by commas. The declaration ends with a semicolon.

Each variable in the declaration can be a symbol, a symbol preceded by an asterisk, or a symbol followed by a (possibly empty) array subscript. Symbols preceded by an asterisk are pointers to a value, rather than the value itself. If the symbol is followed by an array subscript, the value between the square brackets defines the number of elements that will be allocated for the array. The number is required for local and global variables, but not for variables passed as a parameter to a function. A symbol with neither an asterisk or an array subscript represents a single value.

Variables defined with the keywords `short` or `int` are two byte signed integers, ranging from -32768 to 32767, or pointers to or arrays of such integers.

Variables defined with the keyword `long` are four byte signed integers, ranging from -2147483648 to 2147483647, or pointers to or arrays of such integers.

Variables defined with the keyword `char` are one byte unsigned integers, ranging from 0 to 255, or pointers to or arrays of such integers. Typically, these variables are used to hold character values, but C does not actually make any distinction between the two types. When used in an expression, a character value is completely equivalent to an integer, and an integer value can be assigned to a character variable. only the least significant byte of an integer is used when assigning its value to a character - overflows are thus prevented.

Pointers always require two bytes of storage, as do `short` and `int` integers. `long` integers require four bytes of storage. Characters require one byte of storage.

Note that arrays are defined by giving the number of elements to reserve space for, not the maximum subscript allowed. In addition, the first element in a C array is numbered 0, and there is no check for subscripts out of range. This leads to a very common and hard to find bug. If an array is declared as

```
int i[10];
```

and then the assignment

```
i[10] = 100;
```

is made, C reports no error, but the two bytes *after* the array have been changed, not the last element in the array! i[9] is the last valid element in the above array.

Variables must be defined before they are used.

Some valid declarations are shown below.

```
char ch, *chptr, charray[101;
int i, *iptr, iarray[101;
short s;
long 1;
```

Function Declarations

A function declaration consists of the function name followed by a list of parameters, separated by commas and enclosed in parenthesis. The parenthesis are required, even if there are no parameters. Next comes the declarations for any parameters used. Each parameter must have a declaration, and there can be no declaration for a variable that is not a parameter.

The header is followed by the function body. The body starts with the { character. This is followed by zero or more local declarations, then by zero or more statements. After all of the statements, the end of the body of the function is marked with a } character.

C parameters are always passed by value. If a function must change the value of a variable, the only way to do so is to pass a pointer to the variable, rather than the variable itself. If a string is used as a parameter to a function, a pointer to the string constant is actually passed, not the string itself.

C functions are fully recursive. Parameters and local variables are unique to a particular call to a function. Keep in mind, however, that global variables are not recursive - changing their value effects all functions!

Each C program must contain a function called `main`. When the C program is executed, execution starts with `main`, regardless of where main appears in the program.

Small-C requires that parameters be defined in the order in which they are listed in the parameter list.

Functions do not need to be defined before they are used. In fact, they do not have to be defined at all - if a function is not defined, Small-C assumes that the link editor will find it later. This is different than full C, which requires that a function appear in a program if it is used. Full C allows for functions to be defined elsewhere by using an `extern` qualifier, which tells the compiler that the function will be resolved later.

Below are some valid C functions. In fact, they form two valid C programs. The first is the shortest legal C program.

```
main() {}

long i;
main()
{for (i = 1; i < 10; ++i)
    {printint (power(2,i));
    putchar ('\r');
    }
}

power (a,b)
long a,b;
(if (b <= 1) return a;
else return power (a,b-l)*a;
}
```

```
printint(i)
int i;
{long den;

if(i)
    {if(i < 0)
        {Putchar('-');
        i = -1;
        }
    den 1000000000;
    while(i < den) den = den/10;
    while(den)
        {putchar(i/den+'O');
        i = i%den;
        den = den/10;
        }
    }
else putchar('O');
}
```

The while Statement

C provides three looping statements. The while statement is used when a repetitive loop is needed that will terminate when a condition is false, and that condition must be tested before the body of the loop is executed for the first time. It is coded as the keyword `while`, followed by an expression enclosed in parenthesis, and then a statement. The expression is first evaluated. If the value of the expression is non-zero, the statement is executed, and the process repeats. If the value of the expression is zero, the statement after the while statement is executed.

Examples:

```
/* an infinite loop */
while (1);

/* an inefficient way to set i to zero */
while (--i);

/* write the string pointed to by str */
i = 0;
while (str[i]) putchar(str[i++]);
```

The do-while Statement

The do-while statement is very much like a while statement, except that the condition is tested at the end of the loop. Thus, the loop body is always executed at least once.

Syntactically, the do-while loop takes the form of a do keyword, followed by a statement, the keyword `while`, an expression enclosed in parenthesis, and a semicolon. The statement is executed, then the expression is evaluated. If the value of the expression is non-zero, the process repeats. If the value of the expression is zero, the statement after the do-while statement is executed.

Examples:

```
/* another way to set i to 0 */
do i--; while (i);

/* raise 2 to the power 10 */
i = 1;
p = 10;
do i = i*2; while (--p);
```

The for Statement

The for statement is used when an iterative loop is needed. The for statement is a bit more flexible than the counterpart in most other high level languages, and also a bit more confusing.

Semantically, the for statement takes the form of the `for` keyword, followed by three expressions enclosed in parenthesis and separated by semicolons. This is all followed by a statement.

When the for statement is encountered, the first expression is evaluated. Next, the second expression is executed. If the result of the second expression is non-zero, the statement and third expression are evaluated (in that order), and the process repeats, starting with the evaluation of the second expression. If the value of the second expression is zero, the statement after the for statement is executed.

The for statement

```
for (exprl; expr2; expr3) stmnt;
```

is actually completely equivalent to

```
exprl;
while (expr2)
    {stmnt;
    expr3;
    }
```

The if Statement

The if statement is used to execute a statement only if a condition is true. It has two forms: with and without an else clause.

With an else clause, the if statement takes the form of the keyword `if`, an expression enclosed in parenthesis, a statement, the keyword `else`, and another statement. The expression is first evaluated. If it is zero, the statement after the `else` is executed, otherwise the statement after the expression is executed.

Without an else clause, the if statement takes the form of the keyword `if`, followed by an expression enclosed in parenthesis, and a statement. The expression is evaluated. If it is non-zero, the statement after the expression is executed, otherwise the statement is skipped.

Examples:

```
/* write the status of a flag */
if (flag) putstring("true");
else putstring("false");

/* call the function doit if i = 47 */
if (i == 47) doit();
```

The switch Statement

The switch statement is basically a computed goto. Syntactically, it is the keyword `switch`, followed by an expression enclosed in parenthesis and a statement. Within the statement (which is generally compound statement)

two new kinds of labels are allowed. The first is the case label. A case label is the keyword `case`, followed by an integer or character constant, a colon, and a statement. The second special label is the default label. it consists of the keyword `default`, followed by a colon and a statement.

When the switch statement is encountered, the expression is evaluated. If any case label has a value that matches the value of the expression, control is transferred to the statement after the case label. If no case label has a value that matches the expression, but the default label is coded, control is transferred to the statement after the default label. If no case label has a matching value and there is no default label, the statement after the expression is skipped.

Two case labels cannot specify the same constant. more than 100 case labels are allowed in any switch statement. Switch statements can contain other switch statements. In that case, case and default labels refer to the innermost switch statement.

Example:

```
switch (i)
    {case 0:
     case 2:
     case 4:
     case 6:
     case 8: print ("even");
              break;
     default:
         switch (i)
             {case 1: case 3: case 5: case7:
                 print ("odd prime");
                 break;
              case 9:
                 print ("odd");
              }
    }
```

The goto Statement

The goto statement allows control to be transferred to some statement other than the one that would normally be executed next.

Syntactically, a goto statement is the keyword `goto`, followed by a label name and a semicolon. Label names follow the same coding rules as identifiers.

The destination for a goto is a label name, followed by a colon and a statement. The label name specified in a goto statement must appear exactly one time as a destination label within the function that contains the goto statement.

Example:

```
main ()
{int i;

i = 0;
top: if (i > 10) goto end;
        printint (i);
        putchar(.\r');
        ++i;
        goto top;
end:;
}
```

The return Statement

The return statement allows a function to return to the function that called it before the end of the function is reached. It also allows a function to return a specific value.

Syntactically, the return statement is the keyword `return`, possibly followed by an expression, and finishing with a semicolon. If the expression is present, it is evaluated, and the value of the function is the value of the expression. If no expression is given, the returned value is not predictable.

Example:

```
main ()
    {printint (power(2,10));}

power (x,n)
int x,n;

    {if (n == 1) return x;
     else return power (x,n-l)*x;
    }
```

break and continue

The break and continue statements allow a loop to be exited, or for the next loop to start, without the use of a goto statement.

Syntactically, the break statement is the keyword `break` followed by a semicolon. It can appear in the statement part of a while, do-while or for loop, or in the statement part of a switch statement. The continue statement is the keyword `continue` followed by a semicolon. It can appear in the statement part of a while, do-while or for loop.

In the four statements that follow, a break would be equivalent to "goto b1;", while a continue is equivalent to "goto c1;".

```
while (test) (stmnt(); cl:;) bl:;

do (stmnt(); cl:;) while (test); bl:;

for (i = 1; i<10; ++i) {stmnt(); cl:;} bl:;

switch (i) {stmnt();} b1:;
```

Compound Statements

Anywhere that a statement is allowed, a compound statement may be used, instead. A compound statement is a {, followed by zero or more statements, and ending with a). Generally, the compound statement is used to cause several statements to be treated as one when determining the size of a loop or the extent of a conditional statement. Examples of compound statements are included throughout chapters 3 and 4.

Expressions

A Small-C expression consists of one or more terms separated by binary operators, optionally preceded by a unary operator. Parenthesis can be used to modify the precedence of operators.

Small-C operators, in order of precedence, are:

Operator	Precedence	Description
-	1	unary minus
!	1	logical negation
~	1	bitwise negation
*	2	integer multiplication
/	2	integer division
%	2	integer modulus
+	3	integer addition
-	3	integer subtraction
<<	4	bit shift left ("multiplies" by powers of 2)
>>	4	bit shift right ("divides" by powers of 2)
<	5	test for less than
>	5	test for greater than
<=	5	test for less than or equal
>=	5	test for greater than or equal
==	6	test for equality
!=	6	test for not equal
&	7	bitwise and
^	8	bitwise exclusive or
|	9	bitwise or
&&	10	logical and
||	11	logical or
=	12	assignment

Table of C Operators

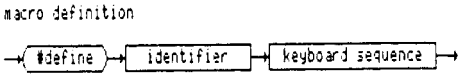
In addition, an l-value may be preceded by one of the following modifiers:

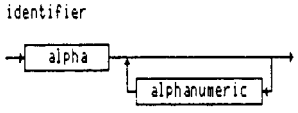
Modifier	Description
*	Causes the value pointed to, rather than the value of a pointer, to be used.
&	The value is the address of the l-value, rather than the l-value.
--	The l-value is decremented, and the decremented value is used in the expression.
++	The l-value is incremented, and the incremented value is used in the expression.

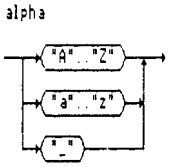
L-values can also be followed by -- or ++. In that case, the l-value is still changed, but the original value is used in the expression.

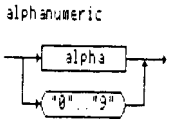
Collected Syntax

The following syntax diagrams summarize the syntax of the Small-C language.

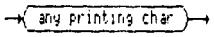






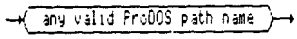


keyboard sequence

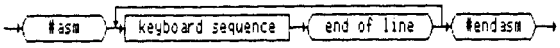


file inclusion

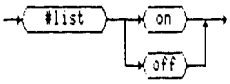
file name



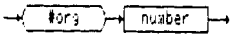
in-line assembly code



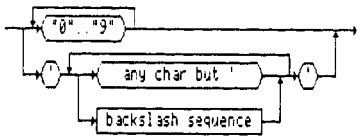
listing control



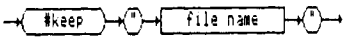
set origin



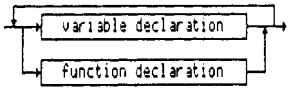
number



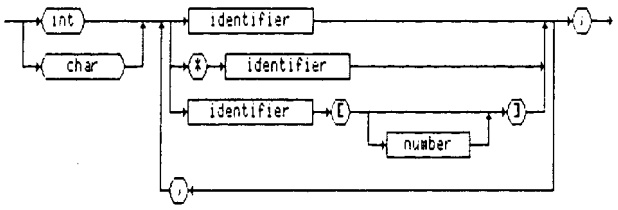
keep output



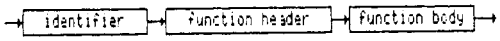
program



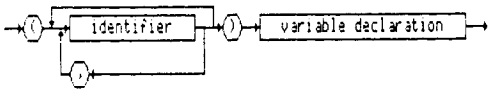
variable declaration



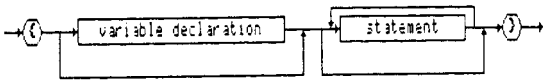
function declaration



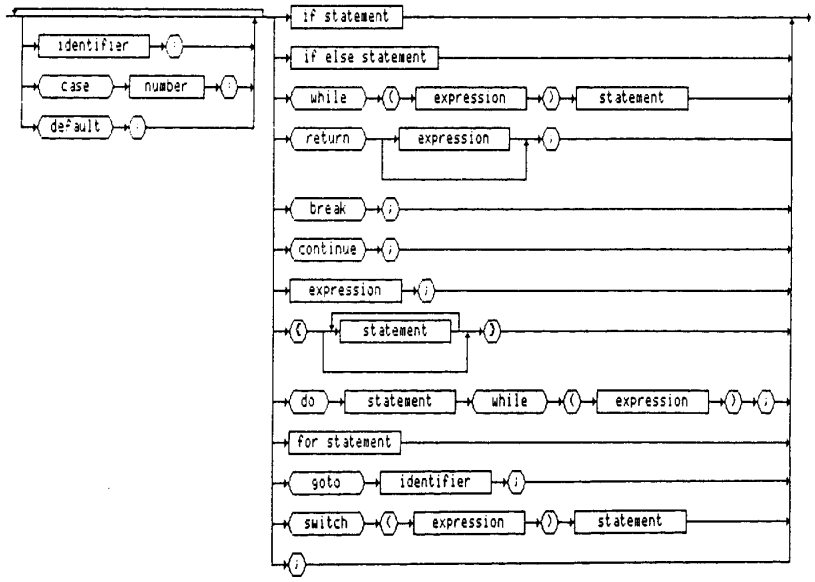
function header



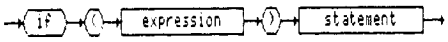
function body



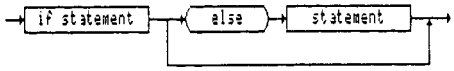
statement



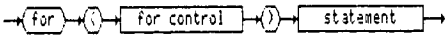
if statement



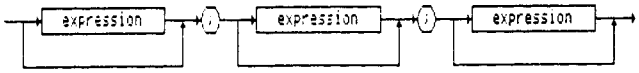
if else statement



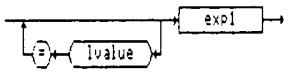
for statement



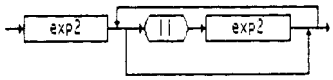
for control



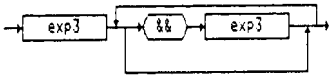
expression



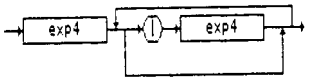
exp1



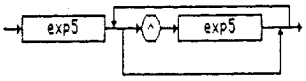
exp2



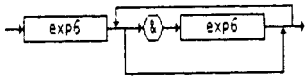
exp3



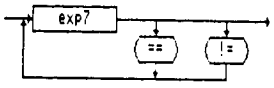
exp4



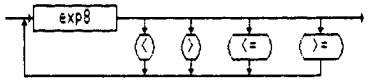
exp5



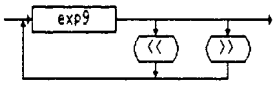
exp6



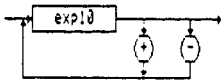
exp7



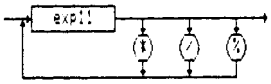
exp8



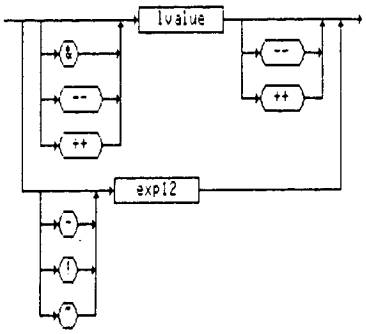
exp9



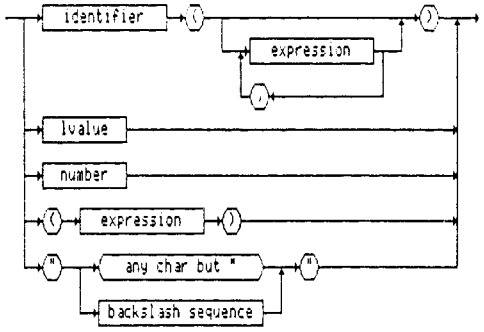
exp10



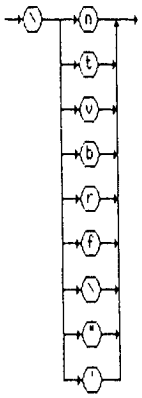
exp11



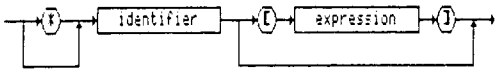
exp12



backslash sequence



lvalue



Libraries

Several of the most important library functions from full C have been provided with Small-C. Basically, those libraries that were helpful for writing the compiler have been included.

fclose (ref)

Close the file whose reference number is `ref`. If `ref` is zero, all open files will be closed.

fgetc (ref)

Returns the next character from the file whose reference number is `ref`. If `ref` is zero, the character is read from the keyboard. If an end of file condition is encountered, or if a read error occurs, -1 is returned.

iscsyn (ch)

`ch` is a character. If it is an alphanumeric character or an underline character, `iscsym` returns a one; otherwise, it returns a zero.

iscsymf (ch)

Returns a one if `ch` is an alphabetic character or the underline character, and a zero if it is not.

isdigit (ch)

Returns a one if `ch` is a numeric character, and a zero if it is not.

fopen (name, code)

`name` is a pointer to a file name, and `code` is a pointer to the kind of open to perform. If `code` points to an "r", the file is opened for reading. If `code` points to a "w", any existing file is deleted and a new file is opened for writing. The function returns a zero if the file cannot be opened, and a reference number for future use by `fclose`, `fqetc` or `fputc` if the file is opened successfully.

Example :

```
ref = fopen ("mydirectory/myfile","r");
```

`fputc (ch, ref)`

`ch` is the character to write. It is written to the file whose reference number is `ref`. If a write error occurs, a -1 is returned; otherwise, `ch` is returned. If `ref` is zero, the character is written to the current output device (usually the CRT).

`putchar (ch)`

This is equivalent to an `fputc (ch, 0)`.

`toupper (ch)`

If `ch` is a lowercase character, `toupper` returns the equivalent uppercase character; otherwise, it returns `ch`.

Chapter 5 - The Compiler

In this chapter, we will look at the Small-C Compiler itself. It is not necessary to read or understand this chapter to use Small-C. The material here is for those who would like to modify the Small-C compiler, or write an ORCA based compiler of their own.

Basic Structure of a Compiler

Almost every modern compiler contains four main parts. While a particular compiler may combine all of the parts into a single program, or may break each of the parts into different passes, there are still four logical ideas being dealt with. To understand a compiler, each of these parts must first be understood.

The first part is the scanner, or lexical analyzer. The scanner is present because compilers deal in symbols, not characters. When a compiler sees the line

```
apples = oranges + 4;
```

it doesn't want to know that the first character is an a, it wants to know that the first thing on the line is a user defined symbol. The scanner is responsible for opening the source file, and reading it character by character. When enough characters have been read to form a token (the technical term for a program symbol), the scanner reports back to the compiler what that token was. In the example above, the scanner would report six tokens:

```
apples
=
oranges
+
4
;
```

The compiler really doesn't care about comments, either, so when the scanner sees the token /*, it skips characters until a */ is found, and doesn't report any of that to the compiler.

The second part of the compiler is the parser. The parser is responsible for insuring that the tokens received from the scanner come in the correct order, and that none are missing. If errors occur, the parser tells you about them. In syntax directed compilers, like Small-C, the parser is in charge. It is the parser that calls the scanner when a token is needed, and the parser that tells the semantic routines that they have something to process.

A good way to think of the parser is as a program that implements the syntax diagrams at the end of the last chapter. In fact, a parser is fairly easy to write from a good set of syntax diagrams.

The next module is the semantic analysis section. In practice, semantic analysis is usually done by a collection of statements and functions mixed in with the parser. This section is responsible for determining the meaning of the program and checking its consistency. For example, noticing that a symbol used in an expression was never defined is the responsibility of semantic analysis.

The last module is the code generator. The code generator is called by the semantic analysis routines. It consists of a collection of subroutines which actually produce the executable program, under the direction of the semantic analysis section.

C is rather unique in that it adds a fifth section to the compiler, called the preprocessor. A preprocessor starts with a source program, modifies it, and writes out another, equivalent source program. While preprocessors are nothing new - Ratfor, for example, is an entire language implemented as a preprocessor - C is the only major language that defines the preprocessor as a required part of the compiler. Logically, the preprocessor isn't really a part of the C language, it is a separate first pass the occurs before the scanner starts building tokens from the input stream. In practice, the preprocessor is often implemented as an integral part of the language. When that is done, as it is in Small-C, the compiler again consists of the four parts mentioned before. The preprocessor, in this case, becomes a part of the scanner.

Intermediate Code, P-code, and Native Code

When the semantic routines need to communicate with the code generator, they often do so through intermediate code. Small-C provides a good example of how this is done. The output from the Small-C compiler is a source file which is then assembled by ORCA/M. In effect, ORCA/M is the

code generator for Small-C. The source file is a file of intermediate code instructions.

If you look at the intermediate code file, you will find a series of very unfamiliar looking instructions. Each of these corresponds to one action requested by the semantic analysis routines. These actions often require several native code instructions to get the result asked for by the semantic analysis routines. For example, the `adi` instruction adds the two integers on the top of the evaluation stack, saving the result back onto the stack. To get this result, the native code macros could expand `adi` to

```
        CLC
        LDY    #0
        LDA    (ESTE),Y
        LDY    # 4
        ADC    (ESTE),Y
        STA    (ESTE),Y
        LDY    #1
        LDA    (ESTE),Y
        LDY    # 5
        ADC    (ESTE),Y
        STA    (ESTE),Y
        LDY    #2
        LDA    (ESTE),Y
        LDY    #6
        ADC    (ESTE),Y
        STA    (ESTE),Y
        LDY    # 3
        LDA    (ESTE),Y
        LDY    #7
        ADC    (ESTE),Y
        STA    (ESTE),Y
        CLC
        LDA    ESTE
        ADC    #4
        STA    ESTE
        BCC    LB1
        INC    ESTE+1
LB1     ANOP
```

This requires 52 bytes of code, and up to 107 CPU cycles! Since the code is so large, it is actually placed into a subroutine and a JSR is inserted in the

code stream. This requires four extra bytes of code and twelve cycles for the first add, but saves forty-nine bytes on each subsequent add.

The idea behind P-code is to write a file that looks more like a CPU designed specifically for the compiler. For example, the `adi` instruction would be implemented as a one byte instruction, saving two bytes over a JSR or fifty-one bytes over true native code. The advantage is obvious: for large programs, the size of the code drops dramatically. Although an interpreter must be included in the binary file, the savings in code size for the compiled program quickly make up for the size of the interpreter. The price paid is a loss in speed while the interpreter loads a byte of P-code, decides which subroutine to call, and does the call.

To summarize, native code compilers produce code that is fast, but can be quite large. P-code compilers generate code that is small, but slower than a native code compiler.

Notice through all of this that only the code generator is different when comparing a P-code and native code compiler. In fact, the Small-C compiler does not know if the finished program will be P-code or native code - that is decided by which macro library is used by the assembler. Thus, Small-C can be set up as a native code compiler when space is not a problem, and as a P-code compiler when space is critical.

Unfortunately, space is critical in the compiler itself. It is so large, in fact, that it is necessary to compile to P-code to get it to fit on an Apple.

Compiling the Compiler

The source code for the Small-C compiler is on the front side of the Small-C Source disk. Depending on what kind of disk drives you have, compiling the compiler can be reasonably straightforward, or a rather delicate experience. You will need at least two floppy disk drives to get the job done.

First we will look at compiling the compiler using large format disk, like a hard disk or a large RAM disk. You will need 600 free blocks to get the job done. Start by moving the source code to a safe spot - another floppy disk will do. Copy the C.PCODE file from the Small-C compiler disk to the same directory as the source, and rename it to be C.MACROS. If the output file should go to another directory, edit CC1. The second line is the `keep` directive. Change it to be the destination path name. Finally, type

```
CMPL CC1
```

If you want a listing, delete the first line in CC1 (the list directive) before starting the compile, and compile with

```
CMPL CC1 >.PRINTER
```

Compiling the compiler with two floppy disk drives is considerably more complicated. Begin by preparing four blank disks. Name these disks /SOURCE, /INTERMEDIATE, /OBJ and /FINAL. Copy all of the source code from the front side of the Small C Source disk to the /SOURCE disk. Add the macro library C.ABSCODE from the main disk and rename it to be C.MACROS. The exact steps to get this done are:

```
<place /SMALL.C in drive 1>
<place /SOURCE in drive 2>
COPY .Dl/C.ABSCODE D2
RENAME .D2/C.ABSCODE .D2/C.MACROS
<place /COMPILER.SRC in drive 1>
COPY .Dl/= .D2
```

Next, enter the editor and change the `keep` directive on the second line of the file CC1 to read

```
#keep "/INTERMEDIATE/CC"
```

The next step is to compile the compiler itself. Start off with your /ORCA disk (with Small-C installed) in drive 1, and the /SOURCE disk in drive 2. Set the prefix to drive two and compile:

```
PREFIX .D2
COMPILE CC1
<place /INTERMEDIATE in drive 1>
```

Don't wait for the system to ask for /INTERMEDIATE before putting it in! As soon as the Small-C compiler clears the screen and prints its prompt, go ahead and swap the disks. There is plenty of time - the compiler must compile all of the global declarations at the start of the program before it need to access the /INTERMEDIATE disk.

It will take quite a while to compile the compiler. When it has finished it is time to assemble the intermediate code file. Start with /SOURCE in drive 1 and /INTERMEDIATE in drive 2. The assembly language portions of the compiler must be copied to /INTERMEDIATE. Finally, the compiler is assembled. The commands are:

```
COPY .D1/CCASM .D2
COPY .D1/CCEQUATES .D2
<place /ORCA in drive 1>
PREFIX .D2
ASSEMBLE CC KEEP=/OBJ/CC
<the assembler asks for /OBJ - put it in drive 1>
<the system asks for /ORCA - put it in drive 1>
<put /OBJ in drive 2>
PREFIX .D2
LINK CC KEEP=/FINAL/CC
<the system asks for /FINAL - put it in drive 1>
<the system asks for /LIBRARY - put it in drive>
<the system asks for /ORCA - put it in drive 1>
```

That completes the compile. Note that the next-to-last step in the list above assumes that your library disk is called /LIBRARY. If not, the system will naturally use the name you have supplied.

Compiling the compiler with more than two floppy disk drives follows the same basic format, but requires fewer disk swaps.

The Small-C Compiler

The Small-C compiler is a fairly large program written mostly in Small-C. A few low level subroutines are written in assembly language. Using assembly language could have been avoided, but using assembly language for a few subroutines increased the speed of the compiler considerably. The source code for the compiler is on the front of the second disk. It is labeled "Compiler Source".

Historically, Small-C was developed by Ron Cain for the 8080 processor, and published in Dr. Dobbs Journal. Ron donated his compiler to the public domain, resulting in several implementations of C based on his effort. This version of Small-C retains much of the flavor of the original, maintaining the. original function names and organization. Changes include installing it

in the ORCA environment, adding the option for p-code, allowing all C statements and most C operations, and supporting long (four byte) integers.

To describe the compiler, its subroutines have been divided into nine categories. Each category will be described, along with the subroutines in that category. Some subroutine descriptions are extremely brief, since their function is so simple. Others are described in more detail. In all cases, it is assumed that a listing of the compiler is available when the descriptions are read.

The first step, then, is to get a listing of the compiler. The most straightforward way of doing this is to compile the compiler and redirect the output to a printer. Compiling the compiler was described in the last section. Another way is to list the individual files using the TYPE command. The files needed are CC1 to CC8, CCASM, and CCEQUATES.

To connect the sections that follow to the previous discussion, the executive, preprocessor and scanner make up the scanner of the compiler. As with most compilers, the parser and semantic analysis are mixed together. They are covered in the sections parse, compiling declarations, compiling functions, compiling statements, and compiling expressions. The code generator is described in the section by that name. The following table summarizes the sections and the subroutines covered in each section.

<u>executive</u>

main	entry point for compiler
ask	get inputs
getlinfo	get input from ORCA.HOST
header	write header
trailer	write trailer
error	write error message
errorsummary	write total error counts
pl	write a line to the CRT
esccheck	check for user exit
formdec	form an integer string
putdec	write an integer
getlnum	get language number

preprocessor

doinclude	handle an include file
readname	read a file name
dolist	handle a list directive
onoff	check for "on" or "off"
dosource	handle a source directive
dokeep	handle a keep directive
doorg	handle an org directive

scanner

inline	get the input line
kill	dispose of remainder of this line
preprocess	expand macros on a line
keepch	keep a preprocessed character
blanks	skip blanks
inchar	read from input file without preprocessing
inbyte	read from input file with preprocessing
ch	get current character
gch	get next character .
nch	get char after this one
number	read a number
pstr	read character constant
qstr	read string constant
strchar	read character from a string
addmac	add a macro definition
putmac	place character in macro table
match	check for string match
streq	compare strings
amatch	check for fixed length string match
astreq	compare fixed length strings
asmstreq	compare string primitive
findglb	search for global label
findloc	search for local label
findmac	search for macro definition

parser

parse	top level of parser
statement	compile a statement
compound	compile a compound statement
ns	insure that a ";" is found
endst	test for end of line

compiling declarations

declglb	process global declarations
declloc	process local declarations
needsub	read size of array
getarg	process function arguments
multidef	flag duplicate symbol error
addglb	add global symbol to symbol table
addloc	add local symbol to symbol table

compiling functions

newfunc	compile a function
skip	skip a function
skiptobracket	skip until a "(" or ")"
search	see if a function should be compiled

compiling statements

doif	compile an if statement
dowhile	compile a while statement
dodo	compile a do statement
dofor	compile a for statement
doreturn	compile a return statement
dobreak	compile a break statement
docont	compile a continue statement
doswitch	compile a switch statement
docase	compile a case statement
dodefault	compile a default label
dogoto	compile a goto statement
dolabel	compile a label
doasm	compile in line assembly code
callfunction	compile a call to a function
needbrack	check for matching ")" or
labelname	check for label name
symname	check for symbol name
getlabel	get next label number
addwhile	add while entry to while table
delwhile	delete while entry from while table
readwhile	get disp to current while entry
test	evaluate a condition

compiling expressions

expression	compile an expression
hierl	compile =
hier2	compile ||
hier3	compile &&
hier4	compile |
hier5	compile ^
hier6	compile &
hier7	compile == !=
hier8	compile <= >= < >
hier9	compile << >>
hier10	compile + -
hier1l	compile * / %
hier12	compile modifiers
hier13	compile variable, array, function references
primary	compile variables and constants
store	save a value
rvalue	load a value
constant	compile constant use

code generation

gen0	generate implied operand instruction
genl	generate one operand instruction
gen2	generate two operand instruction
optimize	check for optimizations
remove	remove instruction from peephole window
instructionout	write an instruction
dumplits	generate string constants
dumpglbs	generate global variables
printlabel	generate a label
outbyte	write a character to the output file
outstr	write a string to the output file
ot	write an op code
outdec	write an integer
comment	write comment character
getmem	load global value
getloc	load address of local value
putmem	save global value
putstk	save local value
indirect	load indirect
defbyte	set up for one byte constant

defstorage	set up for storage reservation
index	index into array
purge	purge code generation buffer

Executive

The executive is a collection of functions that have little to do with the actual compilation process. These functions are concerned with the nuts and bolts of interfacing the compiler to the outside world.

`main - entry point for the compiler`

`main` initializes the global variables, gets the input by calling ask, compiles the program by calling `parse`, and cleans up after the compile is finished.

`ask - get inputs`

`ask` clears the input line so that the first character fetch by the scanner will cause a line to be read. It then opens the first input file and counts the number of functions in the parm list. The parm list is the list of functions for a partial compile.

`getlinfo - get input from ORCA.HOST`

This assembly language subroutine gets the information about the program to be compiled from ORCA.HOST and places it in several global variables, where the information can be accessed by the C portion of the compiler.

`header - write header`

Clears the screen and writes the compiler name to the CRT.

`trailer - write trailer`

After the compile is complete, `trailer` is called to write the last end directive to the output file. if there is more source code (from an assembly language file), an append directive to the next file is also written.

`error - write error message`

Writes the line that caused the error, followed by the error message. Note that the source line has been processed by the preprocessor before `error` writes it. This means that all macros will be expanded.

`errorsummary - write total error count`

Writes the number of errors flagged during the compile. In Small-C, all errors have an error level of 32, indicating that recompilation is necessary after correcting the error.

`pl - write a line to the CRT`

Writes a zero terminated string to the CRT, following it with a carriage return.

`esecheck - check for user exit`

Before writing a line, the system checks to see if the last key pressed was an ESC, indicating that an early termination is desired. If so, an early end of file is created, stopping the compiler.

`formdec - form an integer string`

There are two functions used to write a number. `putdec` writes the number to the CRT, while `outdec` writes a number to the intermediate code file. Both functions call `formdec` to turn the binary number into a ASCII string.

`putdec - write an integer`

Writes an integer to the CRT.

`getlnum - get language number`

Returns the language number of a file, allowing

Small-C to verify that an included file is a Small-C file, as opposed to an assembly language file.

Preprocessor

This section describes the subroutine that carry out the functions of the preprocessor. Parts of the preprocessor are also described in the scanner section that follows.

doinclude - handle an include file

readname is called to read the file name and make sure that it is valid. Next, the language number is checked to make sure the file is a Small-C file. If not, an append is generated at the end of the output file and compilation is stopped. Finally, the included file is opened.

readname - read a file name

A file is read from the current input line. If an error is encountered, zero is returned, otherwise the length of the file name is returned.

dolist - handle a list directive

listflag controls whether or not the source file is listed. The flag is set if "on" is found on the source file, and cleared otherwise.

onoff - check for "on" or "off"

Reads an "on" or "off" keyword from the input line, returning true (1) if "on", and false (0) otherwise.

dosource - handle a source directive

`ctext` is set, determining if the source file is written to the intermediate code file.

dokeep - handle a keep directive

It is an error if the keep directive appears twice, or if it appears after a function. If neither error occurs, a file name is read, and saved for later use by the assembly language routines.

doorg - handle an org directive

Reads an ORG location from the input line and writes an org directive to the intermediate code file, where the assembler will handle it in the normal way.

Scanner

The scanner reads in a line and preprocesses it, replacing macro references by the proper expansion. Primitives are provided to skip blanks, fetch characters, search for symbols, and to read constants from the file.

Basically, lines are read by `inline`, and macros expanded by `preprocess`. The result is stored in an internal line buffer called `line`. It is the line in this internal buffer that is compiled.

inline - get the input line

`inline` reads the next line from the input file and places it in the `line` buffer. If `listflaq` is true (indicating that a source listing is desired), `eof` is false (indicating that this read did not hit an end of file mark) and `skipping` is false (indicating that the current function is being compiled) the line is listed. If there is an output file (cmode == true) and the source is being sent to it (ctext == true) the line is also sent to the output file.

kill - dispose of remainder of this line

Marks the current line as empty so that a subsequent character request reads in a new line.

preprocess - expand macros in a line

The line in the current input buffer is scanned. Extra blanks are removed. Character and string constants are copied intact. Comments are removed, reading in new lines to get to the end of the comment, if that is necessary. Symbol names are truncated to eight characters. If a symbol is the name of a macro, the characters defined by the define directive are placed in the output line, instead of the macro name. Finally, the line is copied back into `line`, and the character pointer set to zero.

keepch - keep a preprocessed character

Saves a character in the preprocessor line buffer.

blanks - skip blanks

Skips blanks and end of line characters.

inchar - read from input file without preprocessing

When `preprocess` is scanning a line to skip comments, it uses `inchar` to fetch the characters. `inchar` will read in a new line if an end of line is found, but unlike the standard character fetch function, `inchar` does not preprocess the line. This avoids an unwanted recursive call to `preprocess`.

inbyte - read from input file with preprocessing

Returns the next character from the input file. If an end of line is found, the next line is read, so this function always returns a character.

ch - get current character

Returns `line (lptr]`, which is the current character. This fun ion has been translated to assembly language to speed up the compiler.

gch - get next character

This function returns the same character that `ch` would return, but it also advances the `lptr` variable.

nch - get character after this one

`nch` implements a one character look-ahead by returning `line (lptr+1),` the character after the one that `ch` would return. This is useful when checking for -two character symbols. For example, the preprocessor uses nch to see if the character after a "/" is "*", signaling the start of a comment.

number - read a number

When a number is found in the input stream, `number` is called to convert it from a string to a binary value.

pstr - read character constant

Reads a character constant from the input stream, returning its value in the `val` parameter.

qstr - read string constant

A string is read from the input file and placed in the literal pool. The literal pool is where all strings are stored. `qstr` returns the location of the string in the `val` parameter.

strchar - read character from a string

`strchar` is called by `pstr` and `qstr` to read a character. It interprets the backslash control codes and checks for the end of the string.

addmac - add a macro definition

Places a macro definition in the macro table. Each macro definition consist of a pair of zero terminated strings. The first is the name of the macro, and the second is the string to substitute when the macro is encountered in the source file.

putmac - place characters in macro table

Places a character at the end of the current macro table.

match - check for a string match

Checks to see if the second string appears at the start of the first string. if so, the string is removed from the input stream and one is returned. if not, a zero is returned.

streq - compare strings

Checks to see if the second string appears at the start of the first. If so, the length of the second string is returned. If not, a zero is returned.

amatch - check for fixed length string match

Checks to see if a fixed length string (the second string parameter) appears at the start of the first string. Additionally, the first string cannot be followed by an alphanumeric character. This allows checking for a keyword, like `return`, without picking up parts of symbols, like the first part of `returned`. If a match is found, a one is returned. Otherwise, a zero is returned.

astreq - compare fixed length strings

Does the string compare for amatch. This function is coded in assembly language for maximum speed.

asmstreg - compare string primitive

This assembly language subroutine is used by `astreq`, `findmac`, `findloc` and `findglb` to see if two strings are equal. It cannot be called from C.

findglb - search for global label

Searches the global symbol table for a symbol. If one is found, a pointer to the symbol is returned, otherwise, a zero is returned. This function is coded in assembly language for maximum speed.

findloc - search for local label

Searches the local symbol table for a symbol. If one is found, a pointer to the symbol is returned, otherwise, a zero is returned. This function is coded in assembly language for maximum speed.

findmac - search for macro definition

Searches the macro pool for a macro. If a matching macro is found, an index into the macro pool `macq` is returned; otherwise, a zero is returned. This function is coded in assembly language for maximum speed.

Parser

Most of the parser functions are mixed with semantic analysis code, but a few are purely for parsing the input. Those are described in this section.

parse - top level of parser

`Parse` has overall control of compiling a program once the compiler has been initialized. It checks to see if a keyword appears, and if so, calls the appropriate subroutine to handle the statement. If no keyword is found, the parser assumes that a function is to be defined, and calls `newfunc`.

statement - compile a statement

`Statement` is called wherever the syntax of C allows a C statement to appear. The syntax is recursive - many C statements can themselves contain statements. With the exception of an expression, all C statements start with a recognizable keyword, so `statement` calls functions to parse individual statements based on the leading keyword. The exception to this is labels although they are not keywords, though, a label is always followed by a colon. `labelname` uses that fact to see if a symbol is a label. If a keyword or label is not found, `expression` is called. After returning from compiling an expression, a pop instruction is generated. This is because C expressions leave the value of the expression on the stack, and it must be removed. The `pop` instruction removes one number from the top of the stack.

compound - compile a compound statement

This function is called by `statement` when a (character is found, indicating the start of a compound statement. `compound` repeatedly calls `statement` until a matching) is found. The `ncmp` counter is used to keep track of how many compound statements are active. If an end of file is encountered and ncmp is not zero, an error is issued.

ns - insure that a ";" is found

Called whenever a ; character is required, `ns` issues an error if the ; is missing. The ; is removed from the input stream.

`endst - test for end of line`

Skips blanks, then checks to see if it is at the end of the file, or if a ; has been found. If either is true, a one is returned; otherwise, a zero is returned.

Compiling Declarations

As variables are declared, they are entered into the symbol table. Global variables and functions are entered into the global symbol table, while local variables and passed parameters are entered into the local symbol table.

Function names are assigned locations by the link editor when the program is linked. Global variables are entered into a common block called SYSCCOM, which is created by the compiler. The compiler also generates a USING SYSCCOM at the start of each function, so all C functions have access to global variables.

Local variables and passed parameters are allocated space on a stack frame. The stack frame for any particular function starts off with a four byte header. The header contains the return address in the first two bytes (used only by the interpreter, and ignored by native code functions), and the address of the start of the calling function's stack frame in the last two bytes. Passed parameters, which always use four bytes of storage, follow the header in the order specified by the function call. Local variables are allocated space after the passed parameters, in the order declared. `char` variables are allocated one byte; `int`, `short` and pointer variables are allocated two bytes; and `long` variables are allocated four bytes.

The evaluation stack appears immediately after any local variables.

`declglb - process global declarations`

The byte or variable being declared is passed in the type parameter. `declqlb` then scans the list of the variables being declared. If the variable name is being preceded by an *, it is changed to a pointer. if it is followed by an array subscript, it is declared as an array. The symbol is entered into the global symbol table by calling addglb.

declloc - process local declarations

Similar to `declglb`, except that the symbol is entered into the local symbol table instead of the global symbol table. sp keeps track of the number of bytes allocated on the stack frame for this function.

needsub - read size of array

If an array is found by `declqlb` or `declloc`, `needsub` is called to find our how long it is. It returns zero if there is no subscript specified, and the number of elements in the array if a number is specified. A reasonable result is always returned, even if the program contains an error.

getarg - process function arguments

Similar to `declloc`, except that variables are only accepted as long as there are unmatched arguments in the parameter list, and four bytes are reserved for a variable, regardless of its type.

The reason that four bytes of space are always reserved is that the parameters are placed on the evaluation stack of the calling function before the current function is called. Since all values on the evaluation stack occupy four bytes of memory, the called function must recognize that fact.

multidef - flay duplicate symbol error

Called to flag duplicate symbol errors.

addglb - add global symbol to symbol table

Enters a new symbol into the global symbol table, and returns a pointer to the new entry.

addloc - add local symbol to symbol table

Enters a new symbol into the local symbol table, and returns a pointer to the new entry.

Compiling Functions

`newfunc - compile a function`

`newfunc` is called to handle anything that `parse` does not recognize. In a correct program, parse recognizes everything except a function declaration.

`newfunc` starts by reading the name of the function, flagging an error if a symbol is not found. It then checks to see if this is a partial compile. If so, `parms[0]` is non-zero. For partial compiles, `newfunc` calls `search` to see if the current function is needed. If not, `skip` is called to skip the function.

Assuming the function is to be compiled, the next step is to see if any functions have been compiled so far. If not, the output file is opened and the first few lines of assembly language are written out.

The function name is now entered into the global symbol table, or, if it is already there, the old entry is checked to make sure it is a function reference. The number of parameters in the parameter list is then counted, and `getarg` is called to declare the passed parameters.

The function body begins with the opening { character. Local declarations are processed by calling `declloc`. After all local declarations have been processed, statements are compiled until a closing } is found.

Clean-up consists of generating a return instruction, deleting all local symbols, purging the peephole optimizer's buffer, writing out string constants, and writing the `end` directive at the end of the function.

`skip - skip a function`

Skips a function during a partial compile by scanning for matching sets of { and } characters.

`skiptobracket - skip until a "{" or "}"`

Used by skip to skip to the next { or } character that is not inside a character constant or string.

`search - see if a function should be compiled`

Searches the list of function names to be compiled during a partial compile to see if the current function is one of the ones that should be compiled.

Compiling Statements

Almost all of the work that the compiler does to understand the meaning of a program and generate code for it is done in this section and the next one, which discusses compiling expressions. The functions in this section can be roughly divided into two groups. The first group is responsible for compiling a particular kind of C statement. Each of those functions starts with the characters "do", and is followed by the name of the C statement that the function compiles. The remaining functions are called to perform specific tasks for the main line routines.

Central to understanding how the looping statements work is a clear understanding of the while queue. The while queue is contained in the array `wq`. The `wq` array functions as a stack, with the topmost element giving information about the structured statement that is currently active. The purpose of this queue is to hold the numbers for two labels. One of these labels is the location to branch to if a `break` statement is found, and the other is the location to branch to if a `continue` statement is found. The statement parsing routine sets up the while queue entry when it is called, assigning numbers to both of these labels at that time. At the appropriate point in the code generation process, the statement parsing routine will also define the two labels, whether or not they have actually been used. Before returning control to the calling function, statement parsing routines delete the top entry on the while queue, so that nested looping statements are supported properly.

Note that only looping statements need to provide branch points for `break` and `continue` statements, so the only functions that use the while queue are those that compile looping statements. These include `dowhile`, `dodo`, `dofor` and `doswitch`.

The while queue is taken directly from the original Small-C compiler, The method is actually poor - better would be to use the natural recursiveness of the C language and eliminate the `wq` array entirely. For one way to do this, examine the way switch statements handle the array of case labels.

doif - compile an if statement

`doif` starts by allocating a label and calling `test` to evaluate the condition. `test` will generate a branch to the label if the condition was false. The statement after the expression is then evaluated with a recursive call to `statement`. An else clause is checked for, and if it is present, an unconditional jump around the second statement is generated, and the statement is compiled.

The code generated for an if statement with an else clause looks like this:

```
          <evaluate condition>
          fjp       sysccl
          <evaluate first statement>
          ujp       syscc2
  sysccl  anop
          <evaluate second statement>
  syscc2  anop
```

If there is no else clause, the code looks like this:

```
  <evaluate condition>
          fjp       sysccl
          <evaluate statement>
  sysccl  anop
```

dowhile - compile a while statement

Labels are assigned for looping and exiting, and then placed in the while queue. The while statement is compiled, and then the while queue entry is deleted. The code generated looks like this:

```
  sysccl  anop
          <evaluate conditiow
          fjp       syscc2
          <evaluate the statement>
          ujp       sysccl
  syscc2  anop
```

dodo - compile a do statement

The do statement is handled much like the while statement. The code generated looks like this:

```
syscc3  anop
        <evaluate the statement>
sysccl  anop
        <evaluate condition>
        tjp       syscc3
syscc2  anop
```

dofor - compile a for statement

Although the `dofor` function is a little longer than those covered so far, there is nothing new here. One point worth noting is the pretzel-like way that the code must be generated due to the poor design of the for statement in C. The for statement is defined in terms of the while statement, with

```
for (expl; exp2; exp3) statement;
```

being equivalent to

```
exp1;
while (exp2)
  {
  statement;
  exp3;
  }
```

Full C compilers are usually designed with a large "putback buffer." This allows the characters in the third expression to be saved until they are really needed, which is after the statement has been compiled. Since Small-C was designed without such a buffer, it is forced to generate a series of jumps to cause the code to execute in the proper order. The code generated looks like this:

```
          <evaluate first expression>
          popi
  syscc1  anop
          <evaluate second expression>
          fjp       sycc2
          ujp       sycc4
  sycc3   anop
          <evaluate third expression>
          popi
          ujp       syccl
  sycc4   anop
          <evaluate the statement>
          ujp       sycc3
  sycc2   anop
```

By adding a putback buffer and saving the third expression until it is needed, the following code could be used instead.

```
          <evaluate first expression>
          popi
  syccl   anop
          <evaluate second expression>
          fjp       sycc2
          <evaluate the statement>
          <evaluate third expression>
          popi
          ujp       syccl
  sycc2   anop
```

doreturn - compile a return statement

The `doreturn` function evaluates an expression if there is one, leaving the value of the expression on the top of the stack. It then generates a `ret` instruction to return to the calling function. The `ret` instruction returns the value that is on the top of the stack as the value of the function.

dobreak - compile a break statement

The break statement simply checks to make sure that there is something in the while queue, indicating that the break statement is inside of a looping statement, and then generates a jump to the exit label for the top while queue entry.

docont - compile a continue statement

This works similarly to `dobreak`, except that the jump is made to the looping label for the enclosing looping statement, instead of to the exit label.

doswitch - compile a switch statement

This is probably the most complicated of all of the statement parsing routines. Part of this is because of the need to make all of the information about the switch statement available outside of the `doswitch` function, so that the `docase` and `dodefault` functions can work properly. At the same time, the `doswitch` function must work recursively with a very large buffer area. It must work recursively so that a switch statement can appear inside of another switch statement. The large buffer is used to save the values for which a case label has been supplied, along with the label number to branch to if that value is encountered.

Basically, the code generated for the switch statement and its accompanying case and default labels starts off with an evaluation of the expression specified in the switch statement itself. A jump around the statement following the expression is then made. Imbedded in this statement are any case and default labels. After the statement part has been evaluated, a jump table is created. The first element in the jump table is the address to jump to if the lowest case label is found, and the last label corresponds to the highest case label specified. Thus, for the switch statement

```
switch(i) {
  case 1: <statement 1>; break;
  case 10: <statement 2>;
  }
```

there would be ten addresses in the jump table.

It is the indexed jump instruction (`xjp`) that makes all of this work. The table of addresses is preceded by the indexed jump instruction. Before the indexed jump is encountered, code has been generated to place a displacement into the jump table on top of the evaluation stack. This is followed by the length of the jump table, so that the indexed jump instruction knows how long the table is. Last, the address of the default jump location is placed on the stack so that the indexed jump instruction

knows where to go if the index is beyond the end of the jump table. The default jump location is the location of the default label if one is specified, or the end of the switch statement if no default label is found.

The code generated for the example shown above would look like this:

```
          <evaluate the control expression>
  ujp     syscc3
  syscc4  anop
          <statement 1>
          ujp       sysccl
  syscc5  anop
          <statement 2>
          ujp       sysccl
  syscc3  anop
          ldci      1
          sbi
          ldci      1
          shl
          ldci      20
          ldci      syscc2
          xjp
          adr       syscc4
          adr       syscc2
          adr       syscc2
          adr       syscc2
          adr       syscc2
          adr       syscc2
          adr       syscc2
          adr       syscc2
          adr       syscc2
          adr       syscc5
  sysccl anop
  syscc2 anop
```

docase - compile a case label

A label for a switch statement is generated, and the label number and the case label value are entered into the case queue. The labels are sorted into ascending order as they are entered.

dodefault - compile a default label

The default label for the enclosing switch statement is generated, and the `founddefault` flag is set to true so that a second default label will not be generated.

dogoto - compile a goto statement

A unconditional branch to the named label is generated. An error is flagged if the label is not a valid symbol, or if the label has been defined as a variable. The peephole optimizer's buffer is purged to force the label to be written out before the `sname` array is deallocated by returning from the function.

dolabel - compile a label

A label is generated, and the peephole optimizer's buffer is flushed to force the name of the label to be used before the `sname` array is deallocated by returning from the function. No error is checked for when the ":" is read, since this function is not called unless `labelname` has verified that the colon was there. `dolabel` ends by recursively calling the `statement` function, fulfilling the C requirement that a label be followed by a statement.

doasm - compile inline assembly code

This function passes all lines in the source code through to the output stream unmodified, until an `#endasm` directive is found. This allows placing assembly code in the middle of a C program.

callfunction - compile a call to a function

`callfunction` is used to generate a call to a function. The code generated starts of with an `mst` instruction, which defines the start of a new stack frame for the function to be called. Next, each of the arguments are evaluated, leaving the value of the expression on the evaluation stack. Finally, a cup instruction is used to call the procedure. The first argument to the cup instruction indicates how long the argument list is. This information is necessary to finish the set up of the new function's stack frame.

needbrack - check for matching ")" or "]"

`needbrack` checks to make sure that the next character is a ")" or "]", whichever was passed as a parameter. If so, it is removed from the input stream. If not, an error is flagged.

labelname - check for label name

Small-C labels must be a valid symbol followed by a colon. This function checks to see if the next token in the input line fits that description, and returns true if it does, and false otherwise.

symname - check for symbol name

This function returns true if the next token is a valid C symbol, and false if it is not. C symbols start with an alphabetic character or the underline character and are followed by zero to seven alphabetic characters, underline characters, and digits.

getlabel - get next label number

Returns the next valid label number.

While this organization is very general, it probably could be improved on by deleting this function and defining the macro

```
#define getlabel ++nxtlab
```

addwhile - add while entry to while table

Places a new entry onto the while queue.

delwhile - delete while entry from while table

Removes the top entry from the while queue.

readwhile - get disp to current while entry

This function returns an index into the while queue. The index points to the top entry currently in the while queue.

`test - evaluate a condition`

This function is called when a C condition is expected. C conditions are expressions enclosed in parenthesis. The function checks for the opening and closing parenthesis, and evaluates the expression. Based on a parameter passed to it, `test` then does a conditional branch (a `tjp` or `fjp`).

Compiling Expressions

Whenever the syntax of C calls for an expression, the `expression` function is called to generate the code necessary to evaluate that expression. The method used is a recursive descent throughout the expression tree. Since C defines an unusual number of precedences between operators, the tree is quite deep.

`expression - compile an expression`

`expression` sets up the `lval` array, used to hold status information about the type of the expression. It then calls `hierl` to evaluate the expression. If the expression evaluated by `hierl` is an `rvalue` (i.e., a variable), code is generated to load that value onto the top of the evaluation stack. For any other type of expression, code to load the value onto the stack has already been generated.

`hierl to hier13 - compile operations`

Most of the work of evaluating an expression is done in the functions whose names start with "hier". Each of these functions is responsible for checking to see if an operator at one level of precedence is present. If so, the part of the expression to the right of the operator is evaluated, followed by generation of the necessary operation. Once the basic idea has been verified, this method of evaluating expressions turns out to be very easy.

`primary - compile variables and constants`

`primary` is called when an actual term in an expression is expected. A term can be a constant, a local variable, a global variable, or a function call. If a symbol is found, but the symbol does not appear in the local or global symbol table, `primary` assumes that it is a function that will be defined later. If so, a parameter list is expected. If the parameter list is not found, an undefined symbol error is flagged.

store - save a value

`store` is called to save the value that is on the top of the evaluation stack. It generates a copy instruction, leaving the value on the stack. This is necessary for support of multiple assignment statements, like

```
a = b = c = 0;
```

rvalue - load a value

`rvalue` is called to load a value from memory, placing it onto the evaluation stack.

constant - compile constant use

Checks to see if the next token is a constant. If so, and if the constant is an integer or a character, a load constant instruction is generated. If the constant is a string, the address of the string is loaded, instead of the string itself.

Code Generation

The code generation routines are called to write instructions to the intermediate code file. It is the intermediate code file which is eventually assembled by the assembler to produce an executable program.

The code generator is built around a peephole optimizer. A peephole optimizer saves instructions as they come in, and looks for short patterns of instructions that can be replaced by equivalent, but shorter, sequences of instructions. `optimize` is the main function for this process: it is `optimize` that is called to scan the list of instructions and replace the unoptimized instructions with optimized instructions.

Central to this theme is the ability to store the instructions being generated in an array, so that the peephole optimizer has a convenient data structure to scan. The instructions are stored in the `peep` array. The size of the window used by the peephole optimizer is controlled by the constant `peepsize`. Making the window larger means that the optimizer can look at more instructions at once, and possibly find optimizations that would otherwise be missed. It also means that the optimizer will take longer.

gen0 - generate implied operand instruction

gen0 is called to generate an instruction that does not have an operand. It calls gen2 with two zero operands.

genl - generate one operand instruction

Generates an instruction with one operand by calling gen2 with a zero for the last operand value.

gen2 - generate two operand instruction

gen2 checks to see if the peephole optimizer's window is already full. If so, the first instruction in the window is written to the output file. The new instruction is placed at the end of the list, and optimize is repeatedly called until no more optimizations are found.

optimize - check for optimizations

optimize checks the peephole window at a location specified as an input parameter for an optimization. The number of optimizations that can be checked for in this way can easily range into the hundreds, so only those that have actually been found in code are checked for. The table below shows the possible optimizations by showing the original code in the left column, and the code that optimize produces in the right column. Note that some optimizations simply rearrange code, making it possible for a later optimization to optimize it better.

```
adi                     ldci   a
ldci   a                adi
adi                     adi

mpi                     ldci   a
ldci   a                mpi
mpi                     mpi

cpoi                    stoi
popi

cpoc                    stoc
popi
```

```
cpol                    stol
popi

croi   a                sroi   a
popi

croc   a                sroc   a
popi

crol   a                srol   a
popi

ldci   a                ldci   a+b
ldci   b
adi

ldci   a                ldci   a-b
ldci   b
sbi

ldci   a                ldci   a*b
ldci   b
mpi

ldci   a                ldci   a/b
ldci   b
dvi

ldci   a                ldci   -a
ngi

ldci   0                (deleted)
adi

ldci   0                (deleted)
sbi

ldci   0                ujp    a
fjp    a

ldci   a                (deleted; a != 0)
fjp    b
```

In addition, any code that appears after a `ujp` or `ret` instruction is removed, unless the code is a label.

`remove - remove instruction from peephole window`

Removes an instruction from the peephole window, moving subsequent instructions forward to fill in the hole.

`instructionout - write an instruction`

Writes an instruction to the output file.

`dumplits - generate string constants`

At the end of each function, `dumplits` is called to generate DC statements to define any string constants used in the function. Each string is followed by a zero .

`dumpglbs - generate global variables`

At the end of the program, `dumpglbs` is called. it generates DS directives to reserve space for each of the global variables. These DS directives appear in a data area called `sysccom`, which can be accessed from all C functions, as well as any assembly language subroutine that issues a USING directive for `sysccom`.

`printlabel - generate a label`

Accepts a label number as an input, and writes the characters "syscc", followed by the label number, to the output file.

`outbyte - write a character to the output file`

Checks to see if an output file is open, and, if so, writes the input character out to the output file.

`outstr - write a string to the output file`

Writes a string to the output file by successive calls to `outbyte`.

ot - write an op code

Writes an operation code to the output file, preceded by a space.

outdec - write an integer

Writes an integer value to the output file.

comment - write comment character

writes a ";" character to the output file.

getmem - load global value

Generates the code to load a value from the global symbol table.

getloc - load address of local value

Generates code to load the address of a value that is on the local stack frame.

putmem - save global value

Generates code to save the value on the top of the stack to a global variable.

putstk - save local value

Generates code to save the value on the top of the stack to a local variable.

indirect - load indirect

Generates an indirect load. An indirect load uses the value on the top of the evaluation stack as the address of the value to load.

defbyte - set up for one byte constant

Writes "dc il" to the output file.

defstorage set up for storage reservation

Writes "ds " to the output file.

`index - index into an array`

The top of the evaluation stack contains an index into an array. If the array is an array of short integers, `index` multiplies the value of the top of the stack by two. If the array is an array of long integers, the value on the top of the stack is multiplied by four.

`purge - purge code generation buffer`

`purge` writes all of the instructions in the code generation buffer out to the output file.

The Intermediate Code Instruction Set

When the Small-C compiler generates code, it does so with its own instruction set. These instructions are written to an output file, and later changed to native 6502 code or P-code using the ORCA/M assembler. Which kind of code is generated depends entirely on the macro library used - Small-C does not know, or care, what kind of code is generated. In fact, simply by changing the macros used, the compiler can generate code for almost any computer.

The following table summarizes the intermediate code instruction set. After the table, instructions are defined in like groups.

The Intermediate Code Instruction Set

```
ldci   1         load a constant
lao    a         load address of local variable
indc             load character, indirect
indi             load integer, indirect
indl             load long integer, indirect
cpoc             copy character, indirect
cpoi             copy integer, indirect
cpol             copy long integer, indirect
stoc             store character, indirect
stoi             store integer, indirect
stol             store long integer, indirect
Idoc   a         load character, absolute
Idoi   a         load integer, absolute
Idol   a         load long integer, absolute
```

```
croc   a         copy character, absolute
croi   a         copy integer, absolute
crol   a         copy long integer, absolute
sroc   a         store character, absolute
sroi   a         store integer, absolute
srol   a         store long integer, absolute
popi             pop TOS
cup    b,a       call user procedure
mst              mark stack
ent    a         set up local stack area
ret              return from procedure
ujp    a         unconditional jump
fjp    a         jump if TOS is false
tjp    a         jump if TOS is true
xjp              indexed jump
adi              +
sbi              -
mpi              *
dvi              /
mod              %
ior              |
xor              ^
andi             &
shr              >>
shl              <<
ngi              unary -
inci             ++
deci             --
eql              ==
neq              !=
les              <
grt              >
leq              <=
geq              >=
ult              unsigned <
ugt              unsigned >
ule              unsigned <=
uge              unsigned >=
not              logical not
ocp              ~ (one's complement)
lor              ||
land             &&
```

The Run Time Stack

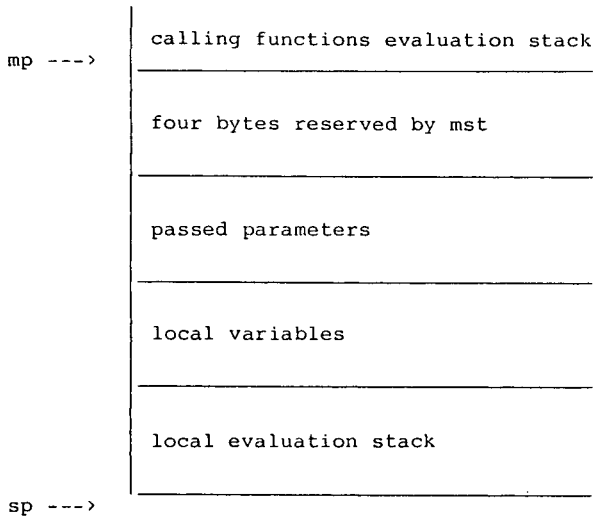
Central to understanding the intermediate code instruction set is a clear understanding of the run time environment. When a Small-C program begins execution, all of free memory minus some space for 1/0 buffers is allocated as a run time stack. The stack starts at the highest available memory location and builds downward.

A function call starts with an `mst`, or mark stack, instruction. The `mst` instruction reserves four bytes on the stack, placing the value of the `mp`, or mark pointer, in two of those bytes. If parameters are being passed to the function being called, the expressions that terminate the parameters are then evaluated. The results of the expressions - which are four bytes each - are left on the stack, so they follow the four bytes reserved by the `mst` instruction.

The next step is to call the function with the `cup` instruction. The first operand is the number of bytes placed on the stack by the `mst` instruction and parameters. This allows the position of the four bytes reserved by `mst` to be computed. The return address is placed there, and the mark pointer, `mp`, is set to that value. Thus, `mp` points to the start of the stack frame for the new function.

When the new function is called, it uses an `ent` instruction to reserve an area on the stack for local variables. The operand for the `ent` instruction is the number of bytes to reserve. Local variables are accessed by computing their address as a distance from the current mark pointer. The area above the local variables is the evaluation stack for the currently executing function.

The `ret` instruction cleans up the stack and returns to the calling function. The mark pointer points to the old mark pointer and return address. These are restored, and replaced with the value or the top of the evaluation stack. The stack pointer is set to point to this area. Note that the end result of this obscure set of operations is that, after the function call, the top of the evaluation stack contains the result of the function. Pictorially, the stack looks like this:



Absolute Loads

Absolute loads know the address of the variable to load, much the same as an absolute load on the 6502. The load comes in three sites: `ldoc`, `ldoi` and `ldol`. These load one byte, two byte, and four byte integers, respectively. In all cases, a four byte number is loaded onto the evaluation stack. One byte values are padded with zeros, while two byte values are sign extended.

Absolute Saves and Copies

As with absolute loads, the operand for an absolute copy or store is the absolute address of the variable to save or copy to. The top number on the evaluation stack is first copied into the variable's location. If the operation is a store, the value is then removed from the stack. Stores and copies still come in three sizes for one, two or four byte operations. Unneeded bytes are truncated without checking for range errors. The absolute copy and store operations are `croc`, `croi`, `crol`, `sroc`, `sroi`, and `srol`.

Indirect Loads

Indirect loads get the address to load from the evaluation stack. The top value on the evaluation stack is the address to load from. It is replaced by

the value loaded. Again, there are three sizes of loads - `indc`, indi `and` `indl` - for one, two, or four byte integers.

Indirect Stores and Copies

The top value on the evaluation stack is the value to save, and the next value on the stack is the address to save to. The value is saved, and the address removed from the stack. In the case of a store, the value is also removed from the stack. The indirect copies and stores are `cpoc`, `cpoi`, `cpol`, `stoc`, `stoi`, and `stol`.

Load Constant

The load constant operation, `ldci`, places a constant value on the top of the evaluation stack. It is used both for numeric constants and fixed addresses.

Load Address

The `lao` instruction is used to load the address of a local variable. The operand is an offset into the current stack frame. This is subtracted from the start location of the current stack frame to get the variable's address, and the address is placed on the evaluation stack.

Pop

The `popi` instruction removes the top value from the evaluation stack.

Jumps

There are four jump instructions available. `ujp` is an unconditional jump that transfers control to the address specified in its operand. `tjp` and `fjp` also have absolute address operands, but they are conditional jumps. Each removes an operand from the evaluation stack. `fjp` will jump if the value is zero, and `tjp` jumps if the value is non-zero.

`xjp` is an indexed jump. It is followed by a table of addresses. When the xjp is encountered, the top three values on the evaluation stack are used. The top most value is the default address. This is where control will be transferred if the jump index is out of range. Next is the length of the jump table, in bytes. An index into the jump table is the last operand. The `xjp` instruction compares the index to the length, and jumps to the default

location if the index exceeds the length. Otherwise, the address to jump to is read from the jump table.

Binary Operators

There is a binary operation for each of the binary operators in the C language. Binary operations remove two values from the top of the evaluation stack, perform an operation on them, and place the result back onto the stack. when the operation starts, the topmost value on the evaluation stack is the second operand; the next value is the first operand.

The binary operations are `adi`, `sbi`, `mpi`, `dvi`, `mod`, `ior`, `xor`, `andi`, `shr`, `shl`, `eql`, `neq`, `les`, `grt`, `leq`, `geq`, `lor`, `land`, `ugr`, `uge`, `ule`, and `ult`. The last four operations are unsigned compares, used for comparing pointers. Note that testing for equality or non-equality is the same for signed or unsigned numbers, so two versions of `eql` and `neq` are not required. The table at the start of the section shows what C operations correspond to what intermediate code operations.

Unary Operators

There is also a unary intermediate code operation for each C unary operator. The unary operations remove one value from the evaluation stack, perform on operation on that value, and place the result back onto the stack.

The unary operators are `ngi`, `inci`, `deci`, and `ocp`.