A 'C' compiler

for

The 8051 microprocessor family

Technical Manual

Release 3.14

Revised 07-Aug-96

1. INTRODUCTION

This complete development package includes all tools and utilities you need to develop 'C' and 'ASM' code for one or more CPU families.

The tools are fully configurable to support virtually any target environment, and most derivatives of the targeted CPU.

1.1 Document conventions

The following conventions are used in this document:

8051 - Indicates the full name of the CPU for which you are developing code.

<text> - Text shown in angle braces indicates data or options that the user MUST supply.

[text] - Text shown in square braces indicates data or options that are optional.

... - Multiple options may be specified.

CPU[,CPU ...]: - Indicates a section or note that applies to only the indicated CPU's.

1.2 Memory Configuration

The runtime library for the compiler can be configured to match the requirements of most target environments. Complete information on the default library setup, and details on customizing the library are contained in a READ.ME file located in the LIB51 subdirectory.

1.3 Code Portability

With few exceptions, this compiler follows the syntax of the "standard" UNIX compiler. Programs written in MICRO-C should compile with few changes under other "standard" compilers.

1.3.1 Unsupported Features:

MICRO-C does not currently support the following features of standard 'C':

*Long / Double / Float / Enumerated data types, Typedef and Bit fields.

* 32 bit "long" number functions are supported via library function calls. These may easily be modified to support larger numbers.

1.3.2 Additional Features

MICRO-C provides a few additional features which are not always included in "standard" 'C' compilers:

Unsigned character variables, Nested comments, 16 bit character constants, Inline assembly code capability.

1.4 Setting up MICRO-C

Once you have installed the files from the distribution

diskettes (See READ.ME on the diskette(s) for instructions), there

are a few simple steps that should be taken to complete

installation of the compiler.

- Setup the MCDIR and TEMP / MCTMP environment variables, as

described in the section entitled "THE COMMAND CO-ORDINATOR".

- Add the MICRO-C home directory to your DOS 'PATH' if you wish

to be able to run the MICRO-C compiler and tools from within other directories on your system.

- Make sure that the runtime libraries are configured correctly for your target system hardware. See the READ.ME file in the LIB51 directory.

Once everything is installed and configured correctly, you are ready to begin using the compiler. There are three ways to run the compiler:

- Use the Integrated Development Environment (DDSIDE), which allows you to enter, compile and debug your programs from a single menu based environment. Complete documentation on DDSIDE is provided in the DDSIDE document.
- Use the "one step" COMMAND CO-ORDINATOR (CC51) to run the compiler from the DOS prompt with a single command. CC51 is described in detail later in this document.
- Run the individual steps of the compiler seperately. This gives you the maximum control of the compiling process, but is not recommended for the novice user. Complete descriptions of the compilation steps and commands are given later in this document.

1.5 The compiling process

There are five programs which work together to completely compile a MICRO-C program:

The PREPROCESSOR (MCP) takes the original 'C' source file, performs MACRO expansion and incorporates the contents of INCLUDE files to get a "pure" 'C' output file. A less powerful pre-processor is also contained inside the COMPILER, which allows this step to be skipped for programs which use only simple

pre-processor functions.

The COMPILER (MCC51) reads a file containing a 'C' source

program, and translates it into an equivalent assembly language

program.

The OPTIMIZER (MCO51) reads the assembly language output from

the compiler identifying and replacing "general" code produced by

the compiler with more efficent code which is equivalent in

specific cases. This step is optional, allowing you to choose

between faster compile time and greater program efficency.

The SOURCE LINKER (SLINK) combines the assembly language output

from the compiler with any required routines from the runtime

library, to create an entire program file.

The ASSEMBLER (ASM51) assembles this program file into a

downloadable binary image in either INTEL or MOTOROLA hex file

format.

2. THE COMMAND CO-ORDINATOR

'CC51' is a utility which co-ordinates the execution of programs

required for each step of the compilation process to provide a simple

"one step" compilation command.

2.1 The CC51 command

The format of the 'CC51' command is:

CC51 <name> [options]

2.1.1 Command line options

-A - produce ASSEMBLER (.ASM) output file

-C - Include 'C' source as comments in ASM files

-F - Fold identical string constants.

-I - Output INTEL hex file (default is MOTOROLA)

-K - KEEP all temporary files (do not delete)

-L - Generate assembly .LST file

-M - Invoke xasm MACRO processor (See XASM manual)

-O - OPTIMIZE the output code (using MCO51)

-P - use the extended PRE-PROCESSOR (MCP)

-Q - QUIET mode (suppress informational messages)

-X - produce eXtended ASM file (includes libraries)

H=mcdir - specify MICRO-C home directory

S=file - specify Startup override file

T=mctmp - specify prefix for TEMPORARY files

name=xx - Predefine macro symbol (use with -P only)

8051: -Z - Zap LJMP/LCALL to AJMP/ACALL (2K addressing)

M=TSCML - Specify the memory model to use:

(Tiny, Small, Compact, Medium, or Large)

When executing the sub-commands, CC51 will search the MICRO-C home directory, as well as any directories specified in the 'PATH' environment variable. Libraries are accessed from the MICRO-C home directory only.

The environment variable 'MCDIR' is examined to determine the path to the MICRO-C home directory. If MCDIR is not defined, CC51 will assume the string '\MC'. You may override this directory by using the 'h=' option on the command line.

Intermediate results from each command are stored in "temporary" files, which are fed as input to the next command. Temporary files will be deleted once they are no longer needed, except in the case where a command fails. When this happens,

any temporary file which was being used as input to that
command will not be deleted, allowing you to examine it for the
cause of the error.

The environment variable 'TEMP' is examined to determine the
directory in which to place temporary files. If you wish to use
a different directory or a special prefix on the temporary file
names, you may override 'TEMP' with the environment variable
'MCTMP'. Note that to allow the use of prefix characters on the
file name, 'MCTMP' is pre-pended to the file name exactly as it
is defined. You may override either directory prefix by using
the 't=' option on the command line.

Here are example 'SET' commands suitable for inclusion in
the AUTOEXEC.BAT file, of an IBM/PC based MICRO-C system which
has the home directory in 'C:\MC', and a TEMP directory on a
RAMDISK as drive 'D':

```
SET MCDIR=C:\MC
SET MCTMP=D:\TEMP\ (Note trailing '\')
```

Note: DO NOT put spaces WITHIN or AFTER the SET command

The '-A' option causes CC51 to bypass the linker and
assembler, and produce an assembly source file (<name>.ASM) as
the output file. If the '-C' option is also used, this file
will contain the 'C' source code in the form of comments. NOTE:
The source code inserted by '-C' will restrict the optimizer to
operate only on sections of code produced by a single 'C' line.

The '-F' option causes the compiler to "fold" its literal
pool (the area of memory where string constants are stored).

This means that identical strings not contained in explicit
variables, will occur only once in memory. Since most programs
never modify such strings, it is usually safe to do this. Note
however that this is a violation of the 'C' standard ("The 'C'
Programming Language" page 181 - "All strings, even when
written identically, are distinct"), and it is possible to
write programs which will not work properly when this option is
used.
The '-X' option is similar to '-A', except that the program
is passed through the source linker, resulting in a ".ASM" file
containing the complete program, including startup code and
library functions.
The '-M' option causes CC51 to invoke the xasm MACRO
processor, allowing macros to be used in your inline assembly
code (See XASM manual). For the sake of efficency, this is done
BEFORE the program is passed through the source linker. This
means that macros in files from the library will NOT be
processed. If library files contain macros, they should be
processed BEFORE they are placed in the library.
The 'S=' option allows you to specify an alternate startup
file (must be in the library) to support the use of multiple
system configurations (see SLINK).

2.2 Troubleshooting

If any of the programs executed by CC51 fail to complete
properly (return a non-zero exit code), CC51 will show a message
"PGM failed (RC)", where PGM is the name of the offending program,

and RC is the exit code value. Common exit code meanings under

MSDOS are:

2 - Command not found (Check MCDIR, PATH)

3 - Invalid directory (Check MCDIR, MCTMP, command line)

4 - Out of file handles (Increase FILES= in CONFIG.SYS)

5 - Bad filename (Check MCDIR, MCTMP)

254 - Program found errors during processing

255 - Program was invoked with incorrect arguments

APPENDIX 1 of the DDSIDE manual contains a complete list of

MSDOS exit codes.

Any time that you have problems getting CC51 to run it's

commands properly, check the settings of the PATH, MCDIR, TEMP and

MCTMP environment variables. Make sure that all directories in

PATH exist and are accessable, and that MCTMP (if used) includes

the trailing '\' to separate the filename from the directory.

(NOTE: When TEMP is used, CC51 automatically adds a '\' if the

environment variable string does not end with one). If your TEMP

specification is unusually long, it may cause CC51 to overrun the

maximum DOS command line length, since it is included multiple

times in some of the commands.

3. THE MICRO-C PROGRAMMING LANGUAGE

The following pages contain a brief summary of the features and

constructs implemented in MICRO-C.

3.1 Constants

The following forms of constants are supported by the compiler:

<num> - Decimal number (0 - 65535)

0<num> - Octal number (0 - 0177777)

0x<num> - Hexidecimal number (0x0 - 0xffff)

'<char>' - Character (1 or 2 chars)

"<string>" - Address of literal string.

The following "special" characters may be used within character

constants or strings:

\n - Newline (line-feed) (0x0a)

\r - Carriage Return (0x0d)

\t - Tab (0x09)

\f - Formfeed (0x0c)

\b - Backspace (0x08)

\<num> - Octal value <num> (Max. three digits)

\x<num> - Hex value <num> (Max. two digits)

\<char> - Protect character <char> from input scanner.

3.2 Symbols

Symbol names may include the characters 'a'-'z', 'A'-'Z',

'0'-'9', and '_'. The characters '0'-'9' may not be used as the

first character in the symbol name. Symbol names may be any

length, however, only the first 15 characters are significant.

The "char" modifier may be used to declare a symbol as an 8 bit

value, otherwise it is assumed to be 16 bits.

eg: char input_char;

The "int" modifier may be used to declare a symbol as a 16 bit

wide value. This is assumed if neither "int" or "char" is given.

eg: int abc;

The "unsigned" modifier may be used to declare a symbol as an

unsigned positive only value. Note that unlike some 'C' compilers,

this modifier may be applied to a character (8 bit) variable.

eg: unsigned char count;

The "extern" modifier causes the compiler to be aware of the existance and type of a global symbol, but not generate a definition for that symbol. This allows the module being compiled to reference a symbol which is defined in another module. This modifier may not be used with local symbols.

eg: extern char getch();

A symbol declared as external may be re-declared as a non-external at a later point in the code, in which case a definition for it will be generated. This allows "extern" to be used to inform the compiler of a function or variable type so that it can be referenced properly before that symbol is actually defined.

The "static" modifier causes global symbols to be available only in the file where they are defined. Variables or functions declared as "static" will not be accessable as "extern" declarations in other object files, nor will they cause conflicts with duplicate names in those files.

eg: static int variable_name;

When applied to local symbols, the "static" modifier causes those variables to be allocated in a reserved area of memory, instead of on the processor stack. This has the effect that the contents of the variable will be retained between calls to the function. It also means that the variable may be initialzed at compile time.

The "register" modifier indicates to the code generator that

this is a high priority variable, and should be kept where it is
easy to get at. See "Functions" for a special use of "register"
when defining a function.

eg: register unsigned count;

8051: See "Memory Allocaation and Referencing" for information
about the 8051 compilers special uses of "register".

Symbols declared with a preceeding '*' are assumed to be 16 bit
pointers to the declared type.

eg: int *pointer_name;

Symbol names declared followed by square brackets are assumed
to be arrays with a number of dimensions equal to the number of
'[]' pairs that follow. The size of each dimension is identified
by a constant value contained within the corresponding square
brackets.

eg: char array_name[5][10];

The "void" modifier is a special case which means this symbol
should never be used as a value. It is usually used to indicate a
function which returns nothing, or a pointer which is never
dereferenced.

eg: void *pointer;

3.2.1 Global Symbols

Symbols declared outside of a function definition are
considered to be global and will have memory permanently
reserved for them. Global symbols are defined by name in the
output file, allowing other modules to access them.

Global variables may be initialized with one or more values, which are expressed as a single array of integers REGUARDLESS of the size and shape of the variable. If more than one value is expressed, '{' and '}' must be used.

eg: int i = 10, j[2][2] = { 1, 2, 3, 4 };

When arrays are declared, a null dimension may be used as the dimension size, in which case the size of the array will default to the number of initialized values.

eg: int array[] = { 1, 2, 3 };

Initialized global variables are automatically saved within the code image, insuring that the initial values will be available at run time. Any non-initialized elements of an array which has been partly initialized will be set to zero.

Non-initialized global variables are not preset in any way, and will be undefined at the beginning of program execution.

3.2.2 Local Symbols

Symbols declared within a function definition are allocated on the stack, and exist only during the execution of the function.

To simplify the allocation and de-allocation of stack space, all local symbols must be declared at the beginning of the function before any code producing statements are encountered.

MICRO-C does not support initialization of non-static local variables in the declaration statement. Since local variables have to be initialized every time the function is entered, you can get the same effect using assignment statements at the beginning of the function.

No type is assumed for arguments to functions. Arguments must be explicitly declared, otherwise they will be undefined within the scope of the function definition.

3.2.3 More Symbol Examples

```
/* Global variables are defined outside of any function */

char a; /* 8 bit signed */

unsigned char b; /* 8 bit unsigned */

int c; /* 16 bit signed */

unsigned int d; /* 16 bit unsigned */

unsigned e; /* also 16 bit unsigned */

extern char f(); /* external function returning char */

static int g; /* 16 bit signed, local to file */

int h[2] = { 1, 2 }; /* initialized array (2 ints) */

main(a, b) /* "int" function containing code */

/* Function arguments are defined between function name and body */

int a; /* 16 bit signed */

unsigned char *b; /* pointer to 8 bit unsigned */

{

/* Local variables are defined inside the function body */

/* Note that in MICRO-C, only "static" locals can be initialized */

unsigned c; /* 16 bit unsigned */

static char d[5]; /* 8 bit signed, reserved in memory */

static int e = 1; /* 16 bit signed, initial value is 1 */

/* Function code goes here ... */

c = 0; /* Initialize 'c' to zero */

strcpy(d, "Name"); /* Initialize 'd' to "name" */

}
```

3.3 Arrays & Pointers

When MICRO-C passes an array to a function, it actually passes

a POINTER to the array. References to arrays which are arguments

are automatically performed through the pointer.

This allows the use of pointers and arrays to be interchangable

through the context of a function call. Ie: An array passed to a

function may be declared and used as a pointer, and a pointer

passed to a function may be declared and used as an array.

3.4 Functions

Functions are essentially initialized global symbols which

contain executable code.

MICRO-C accepts any valid value as a function reference,

allowing some rather unique (although non-standard) function

calls.

For example:

```
function(); /* call function */
variable(); /* call contents of a variable */
(*var)(); /* call indirect through variable */
(*var[x])(); /* call indirect through indexed array */
0x5000(); /* call address 0x5000 */
```

MICRO-C accepts both the "classic" and "modern" formats of

argument definition for a function.

In the "classic" format, only the argument names are placed in

the brackets following the function name. The closing bracket is

followed by formal declarations for the arguments (in any order):

eg: int function(a, b, c) int a, c; char b; { ... }

If the "modern" format, complete declarations for EACH argument are enclosed in the brackets following the function name:

eg: int function(int a, char b, int c) { ... }

Since this is a single pass compiler, operands to functions are evaluated and pushed on the stack in the order in which they are encountered, leaving the last operand closest to the top of the stack. This is the opposite order from which many other 'C' compilers push operands.

For functions with a fixed number of arguments, the order of which operands are passed is of no importance, because the compiler looks after generating the proper stack addresses to reference variables. HOWEVER, functions which use a variable number of arguments are affected for two reasons:

1) The location of the LAST arguments are known (as fixed offsets from the stack pointer) instead of the FIRST.

2) The symbols defined as arguments in the function definition represent the LAST arguments instead of the FIRST.

If a function is declared as "register", it serves a special purpose and causes the accumulator to be loaded with the number of arguments passed whenever the function is called. This allows the function to know how many arguments were passed and therefore determine the location of the first argument.

3.5 Structures & Unions

Combinations of other variable types can be organized into STRUCTURES or UNIONS, which allow them to be manipulated as a single entity.

In a structure, the individual items occur sequentially in memory, and the total size of the structure is the sum of its elements. Structures are usually used to create "records", in which related items are grouped together. An array of structures is the common method of implementing an "in-memory" database.

A union is similar to a structure, except that the individual items are overlaid in memory, and the total size of the union is the size of its largest element. Unions are usually used to allow a single block of memory to be accessed as different 'C' variable types. An example of this would be in handling a message received in memory, in which a "type" byte indicates how the remainder of the message data should be interpreted.

Here are some example of how structures are defined and used (unions are defined and used in an identical manner, except that the word 'union' is substituted for 'struct'):

```
/* Create structure named 'data' with 'a,b,c & d' as members */
struct {
int a;
int b;
char c;
char d; } data;
/* Create structure template named 'mystruc'... */
/* No actual structure variable is defined */
struct mystruc {
int a;
int b;
char c;
char d; };
/* Create structure named 'data' using above template */
```

```
struct mystruc data;

/* Create structure template 'mystruc', AND define a */

/* structure variable named 'data' */

struct mystruc {

int a;

int b;

char c;

char d; } data;

/* Create an array of structures, a pointer to a structure */

/* and an array of pointers to a structure *

struct mystruct array[10], *pointer, *parray[10];
```

```
/* To set value in structure variable/members */

data.a = 10; /* Direct access */

array[1].b = 10; /* Direct array access */

pointer->c = 'a'; /* Pointer access */

parray[2]->d = 'b'; /* Pointer array access */

/* To read value in structure variable/members */

value = data.a; /* Direct access */

value = data[1].b; /* Direct array access */

value = pointer->c; /* Pointer access */

value = parray[2]->d; /* Pointer array access */
```

3.5.1 Notes on MICRO-C structure implementation:

Structures and Unions as implemented in MICRO-C are similar

to the implementation of the original UNIX 'C' compiler, and

are bound by similar limitations, as well as a few MICRO-C

specific ones. Here is a list of major differences when compared to a modern ANSI compiler:

All structure and member names MUST be unique within the scope of the definition. A special case exists, where common member names may be used in multiple structure templates if they have EXACTLY the same type and offset into the structure. This also saves symbol table memory, since only one copy of the member definition is actually kept.

MICRO-C does NOT pass entire structures to functions on the stack. Like arrays, MICRO-C passes structures by ADDRESS. Structure variables which are function arguments are accessed through pointers. For source code compatibility with compilers which do pass the entire structure, if you declare the argument as a direct (non-pointer) structure, the direct ('.') operator is used to dereference it, even though it is actually a pointer reference.

If you MUST have a local copy of the structure, use something like:

```
func(sptr)
struct mystruc *sptr;
{
struct mystruc data;
memcpy(data, sptr, sizeof(data));
...
}
```

To obtain the size of a structure from its template name, use the 'struct' keyword in conjunction with the 'sizeof' operator. In the above example, you could replace 'sizeof(data)' with 'sizeof(struct mystruc)'.

To obtain the size of a structure member, you must specify

it in the context of a structure reference with another symbol:

sizeof(variable.member) or sizeof(variable->member)

NOTE: The current compiler allows almost any symbol to the

left of the '.' or '->' operator in a 'sizeof', however future

versions of the compiler may insist on a structure variable or

a pointer to structure variable.

MICRO-C is quite limited in its implementation of pointers

to structures. Such pointers are internally stored as pointers

to 'char', and therefore dereferencing (*), indexing ([]), and

all forms of pointer arithmetic (++, --, +, -, ...) will

generally not perform as you would expect them to with

structures. The only meaningful things that can be done with a

pointer to a structure is assign it to another variable, pass

it as a function argument and apply the '->' operator to access

individual members.

Use this to "increment" a pointer to a structure to point to

the next structure:

```
ptr += sizeof(struct mystruc)
```

Use this to access the 'n'th structure from the pointer:

```
(ptr + (n * sizeof(struct mystruc)))->member
```

The 'struct' and 'union' keywords are not accepted in a

TYPECAST. This is most commonly used to setup a pointer to a

structure. Since MICRO-C stores its pointers to structures as

pointers to char, you can use (char*) as the typecast, and get

the same functionality.

A structure name by itself (without '.member') acts in a

manner similar to a character array name. With no operation,

the address of the structure is returned. You can also use '[]'

to access individual bytes in the structure by indexing,

although doing so is highly non-portable.

MICRO-C allows static or global structures to be initialized

in the declaration, however the initial values are read as an

array of bytes, and are assigned directly to structure memory

without regard for the type or size of its members:

```
struct mystruc data = { 0, 1, 0, 2, 3, 4 };
/* A=0:1, B=0:2, C=3, D=4 */
```

You can use INT and CHAR to switch back and forth between

word/byte initialization within the value list:

```
struct mystruct data = { int 0, 1, char 2, 3 };
/* A=0, B=1, C=2, D=3 */
```

Strings encountered during structure initializations will be

encoded as a series of bytes if byte initialization (char) is

in effect, or as pointers into the literal pool if word

initialization (int) is in effect.

MICRO-C does not WORD ALIGN structure elements. When using a

processor which requires word alignment, it is the programmers

responsibility to maintain alignment, using filler bytes etc.

when necessary.

3.6 Control Statements

The following control statements are implemented in MICRO-C:

```
if(expression)
```

```
statement;

if(expression)

statement;

else

statement;

while(expression)

statement;

do

statement;

while expression;

for(expression; expression; expression)

statement;

return;

return expression;

break;

continue;

switch(expression) {

case constant_expression :

statement;

...

break;

case constant_expression :

statement;

...

break;

.

.

.
```

default:

statement; }

label: statement;

goto label;

asm "...";

asm {

...

}

3.6.1 Notes on Control Structures

1) Any "statement" may be a single statement or a compound

statement enclosed within '{' and '}'.

2) All three "expression"s in the "for" command are optional.

3) If a "case" selection does not end with "break;", it will

"fall through" and execute the following case as well.

4) Expressions following 'return' and 'do/while' do not have

to be contained in brackets (although this is permitted).

5) Label names may preceed any statement, and must be any

valid symbol name, followed IMMEDIATELY by ':' (No spaces

are allowed). Labels are considered LOCAL to a function

definition and will only be accessable within the scope

of that function.

6) The 'asm' statement used to implement the inline assembly

language capability of MICRO-C accepts two forms:

asm "..."; <- Assemble single line.

asm { <- Assemble multiple lines.

...

}

3.7 Expression Operators

The following expression operators are implemented in MICRO-C:

3.7.1 Unary Operators

- - Negate

~ - Bitwise Complement

! - Logical complement

++ - Pre or Post increment

-- - Pre or post decrement

* - Indirection

& - Address of

sizeof - Size of a object or type

(type) - Typecast

3.7.2 Binary Operators

+ - Addition

- - Subtraction

* - Multiplication

/ - Division

% - Modulus

& - Bitwise AND

| - Bitwise OR

^ - Bitwise EXCLUSIVE OR

<< - Shift left

>> - Shift right

== - Test for equality

!= - Test for inequality

> - Test for greater than

< - Test for less than

>= - Test for greater than or equal to

<= - Test for less than or equal to

&& - Logical AND

|| - Logical OR

= - Assignment

+= - Add to self assignment

-= - Subtract from self assignment

*= - Multiply by self assignment

/= - Divide by and reassign assignment

%= - Modular self assignment

&= - AND with self assignment

|= - OR with self assignment

^= - EXCLUSIVE OR with self assignment

<<= - Shift left self assignment

>>= - Shift right self assignment

NOTES:

1) The expression "a && b" returns 0 if "a" is zero, otherwise the

value of "b" is returned. The "b" operand is NOT evaluated if

"a" is zero.

2) The expression "a || b" returns the value of "a" if it is not 0,

otherwise the value of "b" is returned. The "b" operand is NOT

evaluated if "a" is non-zero.

3.7.3 Other Operators

; - Ends a statement.

, - Allows several expressions in one statement.

+ Separates symbol names in multiple declarations.

+ Separates constants in multi-value initialization.

+ Separates operands in function calls.

? - Conditional expression (ternary operator).

: - Delimits labels, ends CASE and separates conditionals.

. - Access a structure member directly.

-> - Access a structure member through a pointer.

{ } - Defines a BLOCK of statements.

( ) - Forces priority in expression, indicates function calls.

[ ] - Indexes arrays. If fewer index values are given than the

number of dimensions which are defined for the array,

the value returned will be a pointer to the appropriate

address.

Eg:

char a[5][2];

a[3] returns address of forth row of two characters.

(remember index's start from zero)

a[3][0] returns the character at index [3][0];

3.8 Inline Assembly Language

Although 'C' is a powerful and flexible language, there are

sometimes instances where a particular operation must be peformed

at the assembly language level. This most often involves either

some processor feature for which there is no corresponding 'C'

operation, or a section of very time critical code.

MICRO-C provides access to assembly language with the 'asm' statement, which has two basic forms. The first is:

```
asm "..." ;
```

In this form, the entire text contained between the double quote characters (") is output as a single line to the assembler. Note that a semicolon is required, just like any other 'C' statement.

Since this is a standard 'C' string, you can use any of the "special" characters, and thus you could output multiple lines by using '\n' within the string. Another important characteristic of it being a string is that it will be protected from pre-processor substitution.

The second form of the 'asm' statement is:

```
asm {
...
}
```

In this form, all lines between '{' and '}' are output to the assembler. Any text following the opening '{' (on the same line) is ignored. Due to the unknown characteristics of the inline assembly code, the closing '}' will only be recognized when it is the first non-whitespace character on a line.

The integral pre-processor will not perform substitution on the inline assembly code, however the external pre-processor (MCP) will substitute in this form. This allows you to create assembly language "macros" using MCP, and have parameters substituted into them when they are expanded:

```
/*
* This macro issues a 'SETB' instruction for its parameter
*/
```

```
#define setbit(bit) asm {\
SETB bit\
}
/*
* This macro WILL NOT WORK, since the 'bit' operand to the SETB
* instruction is contained within a string and is therefore
* protected from substitution by the pre-processor
*/
#define setbit(bit) asm " SETB bit";
```

3.9 Preprocessor Commands

The MICRO-C compiler supports the following pre-processor commands. These commands are recognized only if they occur at the beginning of the input line.

NOTE: This describes the limited pre-processor which is integral to the compiler, see also the section on the more powerful external processor (MCP).

3.9.1 #define <name> <replacement_text>

The "#define" command allows a global name to be defined, which will be replaced with the indicated text whenever it is encountered in the input file. This occurs prior to processing by the compiler.

3.9.2 #file <filename>

Sets the filename of the currently processing file to the given string. This command is used by the external pre-processor (MCP) to insure that error messages indicate the original source file.

3.9.3 #include <filename>

This command causes the indicated file to be opened and read in as the source text. When the end of the new file is encountered, processing will continue with the line following "#include" in the original file.

3.9.4 #ifdef <name>

Processes the following lines (up to #else or #endif) only if the given name is defined.

3.9.5 #ifndef <name>

Processes the following lines (up to #else or #endif) only if the given name is NOT defined.

3.9.6 #else

Processes the following lines (up to #endif) only if the preceeding #ifdef or #ifndef was false.

3.9.7 #endif

Terminates #ifdef and #ifndef

NOTE: The integral pre-processor does not support nesting of the #ifdef and #idndef constructs. If you wish to nest these conditionals, you must use the external pre-processor (MCP).

3.10 Error Messages

When MICRO-C detects an error, it outputs an informational message indicating the type of problem encountered.

The error message is preceeded by the filename and line number where the error occured:

```
program.c(5): Syntax error
```

In the above example, the error occured in the file "program.c" at line 5.

The following error messages are produced by the compiler:

3.10.1 Compilation aborted

The preceeding error was so severe than the compiler cannot proceed.

3.10.2 Constant expression required

The compiler requires a constant expression which can be evaluated at compile time (ie: no variables).

3.10.3 Declaration must preceed code.

All local variables must be defined at the beginning of the function, before any code producing statements are processed.

3.10.4 Dimension table exhausted

The compiler has encountered more active array dimensions than it can handle.

3.10.5 Duplicate local: 'name'

You have declared the named local symbol more than once within the same function definition.

3.10.6 Duplicate global: 'name'

You have declared the named global symbol more than once.

3.10.7 Expected '<token>'

The compiler was expecting the given token, but found something else.

3.10.8 Expression stack overflow

The compiler has found a more complicated expression than it can handle. Check that it is of correct syntax, and if so, break it up into two simpler expressions.

3.10.9 Expression stack underflow

The compiler has made an error in parsing the expression.

Check that it is of correct syntax.

3.10.10 Illegal indirection

You have attempted to perform an indirect operation ('*' or

'[]') on an entity which is not a pointer or array. This error

will also result if you attempt to index an array with more

indices than it has dimensions.

3.10.11 Illegal initialization

Local variables may not be initialized in the declaration

statement. Use assignments at the beginning of the function

code to perform the initialization.

3.10.12 Illegal nested function

You may not declare a function within the definition of

another function.

3.10.13 Illegal pointer operation

You are attempting to perform an operation which is not

allowed in pointer arithmetic.

3.10.14 Improper type of symbol: 'name'

The named symbol is not of the correct type for the

operation that you are attempting. Eg: 'goto' where the symbol

is not a label.

3.10.15 Improper #else/#endif

A #else or #endif statement is out of place.

3.10.16 Inconsistant member type/offset: 'name'

The named structure member is multiply defined, and has a

different type, offset or dimension than its first definition.

3.10.17 Inconsistant re-declaration: 'name'

You have attempted to redefine the named external symbol

with a type which does not match its previously declared type.

3.10.18 Incorrect declaration

A statement occuring outside of a function definition is not

a valid declaration for a function or global variable.

3.10.19 Invalid '&' operation

You have attempted to reference the address of something

that has no address. This error also occurs when you attempt to

take the address of an array without giving it a full set of

indicies. Since the address is already returned in this case,

simply drop the '&'. (The error occurs because you are trying

to take the address of an address).

3.10.20 Macro expansion too deep

The compiler has encountered a nested macro reference which

is too deep to be resolved.

3.10.21 Macro space exhausted

The compiler has encountered more macro ("#define") text

than it has room to store. Use the external MCP pre-processor

which has much greater macro storage capability.

3.10.22 No active loop

A "continue" or "break" statement was encountered when no

loop is active.

3.10.23 No active switch

A "case" or "default" statement was encountered when no

"switch" statement is active.

3.10.24 Not an argument: 'name'

You have declared the named variable as an argument, but it

does not appear in the argument list.

3.10.25 Non-assignable

You have attempted an operation which results in assignment

of a value to an entity which cannot be assigned. (eg: 1 = 2);

8051: In TINY model, non-register globals are "non-assignable".

3.10.26 Numeric constant required

The compiler requires a constant expression which returns a

simple numeric value.

3.10.27 String space exhausted

The compiler has encountered more literal strings than it

has room store.

3.10.28 Symbol table full

The compiler has encountered more symbol definitions than it

can handle.

3.10.29 Syntax error

The statement shown does not follow syntax rules and cannot

be parsed.

3.10.30 Too many active cases

The compiler has run out of space for storing switch/case

tables. Reduce the number of active "cases".

3.10.31 Too many defines

The compiler has encountered more '#define' statements than

it can handle. Reduce the number of #defines.

3.10.32 Too many errors

The compiler is aborting because of excessive errors.

3.10.33 Too many includes

The compiler has encountered more nested "#include" files

than it can handle.

3.10.34 Too many initializers

You have specified more initialization values than there are

locations in the global variable.

3.10.35 Too many pointer levels

You have declared an item with more levels of re-direction

('*'s) than the compiler can handle.

3.10.36 Type clash

You have attempted to use a value in a manner which is

inconsistant with its typing information. Also results from an

attempt to declare a non-pointer variable with the "void" type.

3.10.37 Unable to open: 'name'

A "#include" command specified the named file, which could

not be opened.

3.10.38 Undefined: 'name'

You have referenced a name which is not defined as a local

or global symbol.

3.10.39 Unknown structure/member: 'name'

You have referenced a structure template or member name

which is not defined.

3.10.40 Unreferenced: 'name'

The named symbol was defined as a local symbol in a

function, but was never used in that function. This error will

occur at the end of the function definition containing the symbol declaration. It is only a warning, and will not cause

the compile to abort.

3.10.41 Unresolved: 'name'

The named symbol was forward referenced (Such as a GOTO

label), and was never defined. This error will occur at the end

of the function definition containing the reference.

3.10.42 Unterminated conditional

The end of file was encountered when a "#if" or "#else"

conditional block was being processed.

3.10.43 Unterminated function

The end of the file was encountered when a function

definition was still open.

3.11 Quirks

Due to its background as a highly compact and portable

compiler, and its target application in embedded systems, MICRO-C

deviates from standard 'C' in some areas. The following is a

summary of the major infractions and quirks:

PLEASE NOTE that this section should not be considered as

evidence that the compiler is somehow inferior or buggy! ALL

compilers have quirks. Most vendors just keep quiet and hope you

won't notice.

3.11.1 Preprocessor quirks

*** NOTE: The quirks in this section apply ONLY to the limited

INTERNAL pre-processor. They DO NOT APPLY when the external

pre-processor (MCP) is used.

When using the INTERNAL pre-processor, the operands to '#'

commands are parsed based on separating spaces, and any portion

of the line not required is ignored. In particular, the

'#define' command only accepts a definition up to the next

space or tab character.

eg: #define APLUSONE A+1 <-- uses "A+1"

#define APLUSONE A +1 <-- uses "A"

Comments are stripped by the token scanner, which occurs

AFTER the '#' commands are processed.

eg: #define NULL /* comment */ <-- uses "/*"

Note that since comments can therefore be included in

"#define" symbols, you can use "/**/" to simulate spaces

between tokens.

eg: #define BYTE unsigned/**/char

Include filenames are not delimited by '""' or '<>' and are

passed to the operating system exactly as entered.

eg: #include \MC\stdio.h

3.11.2 Function/Variable declaration quirks

The appearance of a variable name in the argument list for

an old style function declaration serves only to identify that

variables location on the stack. MICRO-C will not define the

variable unless it is explicitly declared (between the argument

list and the main function body). In other words, all arguments

to a function must be explicitly declared.

MICRO-C does not support "complex" declarations which use

brackets '()' for other than function parameters. These are

most often used in establishing pointers to functions:

int (*a)(); /* Pointer to function returning INT */

(*a)(); /* Call address in 'a' */

Since MICRO-C allows you to call any value by following it

with '()', you can get the desired effect in the above case, by

declaring 'a' as a simple pointer to int, and calling it with

the same syntax:

```
int *a; /* Pointer to INT */
(*a)(); /* Call address in 'a' */
```

MICRO-C will not output external declarations to the output

file for any variables or functions which are declared as

"extern", unless that symbol is actually referenced in the 'C'

source code. This prevents "extern" declarations in system

header files (such as "stdio.h") which are used as prototypes

for some library functions from causing those functions to be

loaded into the object file. Therefore, any "extern" symbols

which are referenced only by inline assembly code must be

declared in the assembly code, not by the MICRO-C "extern"

statement.

Read the notes at the end of the section entitled

"Structures and Unions" for information on limitations or

differences from standard 'C' in MICRO-C's implementation of

structures and unions.

3.11.3 Operator/Expression quirks

MICRO-C is more strict about its handling of the ADDRESS

operator ('&') than most other compilers. It will produce an

error message if you attempt to take the address of a value

which is already a fixed address (such as an array name without

a full set of indicies). Since an address is already produced in such cases, simply drop the '&'.

The INDEXING operator '[]' is not commutative in MICRO-C. In

other words 'array[index]' is NOT equivalent to 'index[array]'.

The 'x' in '0x' and '\x' is accepted in lower case only.

When operating on pointers, MICRO-C only scales the

increment (++), decrement (--) and index ([]) operations to

account for the size of the pointer:

```
eg: char *cptr; /* pointer to character */
int *iptr; /* pointer to integer */
++cptr; /* Advance one character */
++iptr; /* Advance one integer */
cptr[10]; /* Access the tenth character */
iptr[10]; /* Access the tenth integer */
cptr += 10; /* Advance 10 characters */
iptr += 10; /* Advance ONLY FIVE integers */
```

NOTE: A portable way to advance "iptr" by integers is:

```
iptr = &iptr[10]; /* Advance 10 integers */
```

Since structures are internally represented as arrays of

"char", incrementing a pointer to a structure will advance only

one (1) byte in memory. To advance to the "next" instance of

the structure, use:

```
ptr += sizeof(struct template);
```

Unlike some 'C' compilers, MICRO-C will process character

expressions using only BYTE values. Character values are not

promoted to INT unless there is an INT value involved in the

expression. This results in much more efficent code when dealing with characters, particularily on small processors

which have limited 16 bit instructions. Consider the statement:

```
return c + 1;
```

On some compilers, this will sign extend the character

variable 'c' into an integer value, and then ADD an integer 1

and return the result. MICRO-C will ADD the character variable

and a character 1, and then promote the result to INT before

returning it (results of expressions as operands to 'return'

are always promoted to int).

Unfortunately, programs have been written which rely on the

automatic promotion of characters to INTs to work properly. The

most common source of problems is code which attempts to treat

CHAR variables as UNSIGNED values (many older compilers did not

support UNSIGNED CHAR). For example:

```
return c & 255;
```

In a compiler which always evaluates character expressions

as INT, the above statement will extract the value of 'c' as

positive integer ranging from 0 to 255.

In MICRO-C, ANDing a character with 255 results in the same

character, which gets promoted to an integer value ranging from

-128 to 127. To force the promotion within the expression, you

could CAST the variable to an INT:

```
return (int)c & 255;
```

The same objective can be achieved in a more efficent (and

correct) manner by declaring the variable 'c' as UNSIGNED CHAR,

or by CASTing the variable to an UNSIGNED value:

```
return (unsigned)c;
```

Note that this is not only more clearly shows the intent of the programmer, but also results is more efficent generated

code.

A related quirk occurs because MICRO-C automatically casts untyped constants used in binary operations to match the other type used in the operation, again this is done to preserve 8 bit arithmetic wherever possible, but it can suprise you if you use a number which cannot be expressed in 8 bits in an operation with a character type. The high bits of the number will be dropped as it is truncated to a matching 8 bit type. If such a number is truly needed in an 8 bit expression, simply cast it to the appropriate 16 bit type:

```
eg: 0x1234 / c; /* 0x34 / c (8 bit arithmetic) */
(int)0x1234 / c; /* 0x1234 / c (16 bit arithmetic) */
```

Most processors do not support a simple efficent method for adding or subtracting a SIGNED 8 bit quantity and a 16 bit quantity. The code generators supplied with MICRO-C make the assumption that character values being added or subtracted to/from integers will contain only POSITIVE values, and thus use UNSIGNED addition/subtraction. This allows much more efficent code to be generated, as the carry/borrow from the low order byte of the operation is simply propagated to the high order byte of the result (an operation supported in hardware by the CPU).

For those rare instances where you do wish to add/subtract a potentially negative character value to/from an int, you can force the expression to be performed in less efficent fully 16 bit arithmetic by casting the character to an int.

```
int i;

char c;

...

i += c; /* Very efficent... C must be positive */

i += (int)c; /* Less efficent... C can be negative */
```

Some of the compiler implementations for simpler 8-bit

processors may not support signed division. In these cases,

unsigned division will be performed in all cases.

4. MEMORY ALLOCATION AND REFERENCING

4.1 8051 Memory Models

The 8051 compiler supports five different "memory models",

which offer various methods of using the 8051 memory architecture:

4.1.1 TINY Model

Assumes NO external RAM is available. Any initialized global

variables are placed in ROM. All references to external memory

are via the CODE address space (MOVC instruction). Local

variables are allocated in INTERNAL ram.

TINY model notes:

- No external RAM required

- Minimal hardware system requirements.

- Only 128/256 bytes RAM available.

- Initialized variables occupy ROM only.

- Initialized variables may not be modified.

4.1.2 SMALL Model

Requires that ROM and RAM be overlapped into a single 64K

address space, which is accessed with MOVX. Local variables are

allocated in INTERNAL ram.

SMALL model notes:

- Up to 64K data + code (combined)
- Requires RAM and ROM to be overlapped in hardware.
- Local variables are fast to access.
- Local variable space is quite limited.
- Initialized variables occupy ROM only.
- Initialized variables may not be modified.

4.1.3 COMPACT Model

Similar to SMALL, except that local variables are allocated

an a "pseudo" stack located in external RAM.

COMPACT model notes:

- Up to 64K data + code (combined)
- Requires RAM and ROM to be overlapped in hardware.
- Local variables may be as large as available RAM.
- Local variables are slower to access.
- Initialized variables occupy ROM only.
- Initialized variables may not be modified.

4.1.4 MEDIUM Model

Similar to SMALL, except that initialized data is copied to

the RAM before program execution begins. This eliminates the

requirement that ROM and RAM be overlapped, and allows

initialized variables to be modified by your program at

runtime.

MEDIUM model notes:

- Allows up to 64K of data + 64K of code & initialized data.
- Does not require overlapped ROM/RAM.
- Local variables are fast to access.
- Local variable space is quite limited.
- Initialized variables occupy RAM.
- Initialized variables may be modified at runtime.

4.1.5 LARGE Model

Similar to COMPACT, except that initialized data is copied to the RAM before program execution begins. This eliminates the requirement that ROM and RAM be overlapped, and allows initialized variables to be modified by you program at runtime.

LARGE model notes:

- Allows up to 64K of data + 64K of code & initialized data.
- Does not require overlapped ROM/RAM.
- Local variables may be as large as available RAM.
- Local variables are slower to access.
- Initialized variables occupy RAM.
- Initialized variables may be modified at runtime.

To differentiate accesses to INTERNAL and EXTERNAL memory, the code generator makes use of the "register" attribute. Any global varables which are declared with the "register" are allocated in INTERNAL memory. Globals which are NOT declared "register" are allocated in EXTERNAL memory. Although this is inconsistant with the standard 'C' use of "register" (which cannot normally be applied to globals), this method was chosen because it preserves the general notion that "register" means "efficent".

4.2 8051 LOCAL Variables

In the TINY, SMALL and MEDIUM memory models, LOCAL variables are allocated on the 8031/8051 internal CPU stack. Since there is only 120 bytes of memory for internal variables and the stack, care should be exercised is the use of local variables when using these memory models. (If you are using an 8032/8052 type processor, the amount of available internal RAM increases to 248 bytes).

In the COMPACT and LARGE memory model, LOCAL variables are allocated in external memory. This allows local variables to be any number and size up to the limits of available RAM.

4.3 8051 GLOBAL Variables

GLOBAL variables which are declared as "register" will be allocated in internal memory. These variables MUST NOT be initialized in the declaration.

GLOBAL variables which are NOT declared as "register" are allocated in EXTERNAL memory. In the TINY, SMALL and COMPACT memory models, any such variables which are initialized in the declaration are allocated in ROM, and thus will not be alterable by your program.

By default, a reference via a pointer occurs to INTERNAL memory if that pointer is declared as "register", and to EXTERNAL memory if it is not.

For GLOBAL pointer variables, this means that the pointer will reference the same type of memory in which it is defined. You may use casts to force a pointer to reference a specific type of memory. For example:

```
/* Global pointer definitions */
register int *iptr; /* Allocated in INTERNAL */
int *eptr; /* Allocated in EXTERNAL */
/* ... pointer accesses (must be inside function) */
*iptr; /* Reference INTERNAL */
*eptr; /* Reference EXTERNAL */
*(int*)iptr; /* Reference EXTERNAL */
*(register int*)eptr; /* Reference INTERNAL */
```

5. ADVANCED TOPICS

This section provides information on the more advanced aspects of MICRO-C, which is generally not needed for casual use of the language.

5.1 Conversion Rules

MICRO-C keep track of the "type" of each value used in all expressions. This type identifies certain characteristics of the value, such as size range (8/16 bits), numeric scope (signed/unsigned), reference (value/pointer) etc.

When an operation is performed on two values which have identical "types", MICRO-C assigns that same "type" to the result.

When the two value "types" involved in an operation are different, MICRO-C calculates the "type" of the result using the following rules:

5.1.1 Size range

If both values are direct (not pointer) references, the result will be 8 bits only if both values were 8 bits. If either value was 16 bits, the result will be 16 bits.

If one value is a pointer, and the other is direct, the
result will be a pointer to the same size value as the original
pointer.
If both values were pointers, the result will be a pointer
to 16 bits only if both original pointers referenced 16 bit
values. If either pointer referenced an 8 bit value, the result
will reference an 8 bit value.

5.1.2 Numeric Scope

The result of an expression is considered to be signed only
if both original values were signed. If either value was an
unsigned value, the result is unsigned.

5.1.3 Reference

If either of the original values was a pointer, the result
will be a pointer. One exception to this rule is the
subtraction of two pointers, which yeilds an integer result.
Note that this "calculated" result type is used for partial
results within an expression. Whenever a symbol such as a variable
or function is referenced, the type of that symbol is taken from
its declaration, no matter what "type" of value was last stored
(variable) or returned (function).
The TYPECAST operation may be used to override the type
calculated for the result of an expression if necessary.

5.2 Accessing on-chip registers and hardware

6808,6811,8051:

You may access any of the on-chip registers directly from with
your 'C' program, by including an appropriate "extern" declaration

for each register prior to using it. The file "8051reg.h" contains
a complete set of declarations for all of the on-chip registers.

5.3 Assembly Language Interface

Assembly language programs may be called from 'C' functions and vice versa. These programs may be in the form of inline assembly language statements in the 'C' source code, or separately linked modules.

The MICRO-C runtime library includes a number of assembly language subroutines which provide various services (such as 16 bit multiplication, division, etc). These routines are well documented in the library startup code files (8051RL?.ASM), which you should examine before attempting to use assembly language within MICRO-C.

Global variables defined in 'C' exist at absolute addresses and may be referenced directly by name from assembly language. Global names which are referenced by both assembly language and 'C' should not be longer than 15 characters.

When MICRO-C calls any routine ('C' or assembler), it first pushes all arguments to the routine onto the processor stack, in the order in which they occur in the argument list to the function. This means that the LAST argument to the function is CLOSEST to the top of the processor stack.

Arguments are always pushed as 16 bit values. Character values are extended to 16 bits, and arrays are passed as 16 bit pointers to the array. (MICRO-C knows that arrays which are arguments are actually pointers, and automatically references through the pointer).

After pushing the arguments, MICRO-C then generates a machine language subroutine call, thereby executing the code of the routine.

Once the called routine returns, the arguments are removed from the stack by the calling program.

It is the responsibility of the called function to remove any saved registers and local variable space from the stack before it returns. If a value is to be returned to the calling program, it is expected to be in the 16 bit ACCUMULATOR.

Examine the supplied library functions, as well as code produced by the compiler to gain more insight into the techniques of accessing local variables and arguments.

8051:

If you modify the 8051 REGISTER BANK in ANY assembly language code (INLINE or FUNCTION), you must reset it to ZERO before your code completes. It is also your responsibility to make sure that you do not "clobber" INTERNAL memory if you move the register bank. If necessary, the startup code can be modified to allow space for additional register banks (This would also be a good idea if you employ interrupt handlers that use the registers).

INLINE code should preserve the contents of R0 (which is used internally for stack addressing). This restriction does not apply to assembly language functions. All other registers may be used freely.

Globals and static variables which are declared as "register" will exist in INTERNAL ram, and may be loaded and stored with

simple instructions. Globals which are not "register" will be in

EXTERNAL ram, and must be accessed through the DATA POINTER (DPTR)

register.

The 8051 processor stack grows UPWARD in memory, which is the

opposite of most other CPU's.

Local variables in a function may be referenced as offsets from

the stack pointer. Note that offsets must be adjusted for the

number of additional bytes which are pushed and popped on the

stack during the execution of the function. The address of a

particular local variable is calculated as:

"stack pointer"

-

(# bytes pushed during function execution)

-

(size of all preceeding local variables in bytes)

Arguments to a function may also be referenced as direct

offsets from the stack pointer, in much the same way as local

variables are. The address of a particular argument is calculated

as:

"stack pointer"

-

(# bytes pushed during function execution)

-

(size of all local variables in bytes)

-

(Size of return address on stack (2))

-

(# arguments from LAST argument) * 2

If a function has been declared as "register", MICRO-C will

load the accumulator with the number of arguments which were

passed, each time the function is called. This allows the function

to determine the location of the first argument, which may be

calculated as:

"stack pointer"

-

(accumulator contents) * 2

-

(# bytes pushed during function execution)

-

(size of all local variables in bytes)

-

(Size of return address on stack (2))

5.4 Compiling for ROM

The output from the compiler is entirely "clean", and may be

placed in Read Only Memory (ROM).

The addresses for code and data storage are established by the

startup files in the runtime library, called 8051RL*.ASM. You may

examine and modify these files to suit your own particular memory

allocation needs.

5.5 Direct/Internal .vs. Extended/External memory

6811,6816,8051:

To allow the user to control allocation of variables in internal and external memory, MICRO-C/11 makes use of the "register" keyword.

GLOBAL variables which are declared as "register" will be allocated in internal memory. These variables MUST NOT be initialized in the declaration. GLOBAL variables which are NOT declared as "register" are allocated in EXTERNAL memory.

To preserve the notion that "register" means "efficent", the 6811 internal memory should be positioned within the first 256 bytes of memory address space, allowing the use of "direct" addressing for those variables.

6. THE MICRO-C PREPROCESSOR

The MICRO-C Preprocessor is a source code filter, which provides greater capabilities than the preprocessor which is integral to the MICRO-C compiler. It has been implemented as a stand alone utility program which processes the source code before it is compiled.

Due to the higher complexity of this preprocessor, it operates slightly slower than the the integral MICRO-C preprocessor. This is mainly due to the fact that it reads each line from the file and then copies it to a new line while performing the macro substitution. This is necessary since each macro may contain parameters which must be replaced "on the fly" when it is referenced.

The integral MICRO-C preprocessor is very FAST, because it does not copy the input line. When it encounters a '#define'd symbol, it simply adjusts the input scanner pointer to point to the definition of that symbol.

Keeping the extended preprocessor as a stand alone utility allows you to choose between greater MACRO capability and faster compilation. It also allows the system to continue to run on very small hardware platforms.

The additional capabilities of the extended preprocessor are:

- Parameterized MACROs.
- Multiple line MACRO's.
- Nested conditionals.
- Ability to undefine MACRO symbols.
- Library reference in include file names.

6.1 The MCP command

The format of the MICRO-C Preprocessor command line is:

```
MCP [input_file] [output_file] [options]
```

[input_file] is the name of the source file containing 'C' statements to read. If no filenames are given, MCP will read from standard input.

[output_file] is the name of the file to which the processed source code is written. If less than two filenames are specified, MCP will write to standard output.

6.1.1 Command Line Options

MCP accepts the following command line [options]:

-c - Instructs MCP to keep comments from the input file (except for those in '#' statements which are always removed). Normally, MCP will remove

all comments.

-l - Causes the output file to contain line numbers.

Each line in the output file will be prefixed

with the line number of the originating line

from the input file.

l=path - Defines the directory path which will be taken

to reference "library" files when '<>' are

used around an '#include' file name. Unless

otherwise specified, the path defaults to:

'\MC'

-q - Instructs MCP to be quiet, and not display the

startup message when it is executed.

<name>=<text> - Pre-defines a non-parameterized macro of the

specified <name> with the string value <text>.

6.2 Preprocesor Commands

The following commands are recognized by the MCP utility, only

if they occur at the beginning of the source file line:

6.2.1 #define <name>(parameters) <replacement text>

Defines a global macro name which will be replaced with the

indicated <replacement text> wherever it occurs in the source

file.

Macro names may be any length, and may contain the

characters 'a'-'z', 'A'-'Z', '0'-'9' and '_'. Names must not

begin with the characters '0'-'9'.

If the macro name is IMMEDIATELY followed by a list of up to

10 parameter names contained in brackets, those parameter names

will be substituted with parameters passed to the macro when it

is referenced. Parameter names follow the same rules as macro

names.

eg: #define min(a, b) (a < b ? a : b)

If any spaces exist between the macro name and the opening

'(', the macro will not be parameterized, and all following

text (including '(' and ')') will be entered into the macro

definition.

If the very last character of a macro definition line is

'\', MCP will continue the definition with the next line (The

'\' is not included). Pre-processor statements included as part

of a macro definition will not be processed by MCP, but will be

passed on and handled by the integral MICRO-C preprocessor.

6.2.2 #undef <symbol>

Undefines the named macro symbol. further references to this

symbol will not be replaced.

NOTE: With MCP, macro definitions operate on a STACK. IE: If

you define a macro symbol, and then re-define it (without

'#undef'ing it first), subsequently '#undef'ing it will cause

it to revert to its previous definition. A second '#undef'

would then cause it to be completely undefined.

6.2.3 #forget <symbol>

Similar to '#undef', except that the symbol and ALL

SUBSEQUENTLY DEFINED SYMBOLS will be undefined.

Useful for releasing any local symbols (used only within a single include file).

For example:

```
#define GLOBAL "xxx" /* first global symbol */
... /* more globals */
#define LOCAL "xxx" /* first local symbol */
... /* more locals */
/* body of include file goes here */
#forget LOCAL /* release locals */
```

6.2.4 #if <expression>

Evaluates an expression, and causes the following lines (up to '#else' or '#endif') to be processed and included in the output file only if the result of the expression was TRUE (non-zero). The expression may contain only numeric constants, and the following operators:

+	Addition	<<	Shift Left
-	Subtract & Unary negate	>>	Shift Right
*	Multiplication	==	Test equal to
/	Division	!=	Test not equal to
%	Modulus	>	Test greater than
&	Bitwise AND	>=	Test greater or equal
|	Bitwise OR	<	Test less than
^	Bitwise XOR	<=	Test less or equal
&&	Logical AND	~	Unary bitwise complement
||	Logical OR	!	Unary logical complement

Note that the simplified expression parser in MCP does NOT follow standard rules of operator precedence. Operations are peformed from left to right in the order that they are encountered. Brackets '()' can be used to force a specific order of evaluation.

Macro substitution will occur on the expression before it is

evaluated, and any symbol names which are not resolved are

assumed to have a value of 0 (FALSE).

6.2.5 #ifdef <symbol>

Causes the following lines (up to '#else' of '#endif') to be

processed and included in the output file only if the named

symbol is defined as a macro.

6.2.6 #ifndef <symbol>

Causes the following lines (up to '#else' of '#endif') to be

processed and included in the output file only if the named

symbol is NOT defined as a macro.

NOTE: '#if/#ifdef/#ifndef#else/#endif' may be nested.

6.2.7 #else

Toggles the state of the "if_flag", controlling conditional

processing. Only has effect in the highest level of suspended

processing. IE: Nested conditionals will work properly.

If the previous '#if/#ifdef/#ifndef' failed, processing will

begin again following the '#else'.

If the previous '#if/#ifdef/#ifndef' passed, processing will

be suspended until the '#endif' is encountered.

NOTE: Since '#else' acts as a toggle, it may be used outside

of any '#if/#ifdef/#ifndef' to unconditionally suspend

processing up to '#endif'. You can also use multiple '#else's

in a single conditional, to swap back and forth between

true/false processing without re-testing the condition.

6.2.8 #endif

Resets the "if_flag" controlling conditionals, causing

processing to resume. Only has effect in the highest level of

suspended processing. IE: Nested conditionals will work

properly.

6.2.9 #include <filename>

Causes MCP to open the named file and include its contents

as part of the input source.

If the filename is contained within "" characters, it will

be opened exactly as specified, and (unless it contains a

directory path) will reference a file in the current directory.

If the filename is contained within the characters '<' and

'>', it will be prefixed with the library path (See 'l='

option), and will therefore reference a file in that library

directory.

For example:

```
#include "header.h" /* from current directory */

#include <stdio.h> /* from library directory */
```

6.2.10 #error <text>

Causes MCP to issue an error message containing the

specified text, and then terminate.

6.3 Error messages

When MCP detects an error during processing of an include file,

it displays an error message, which is preceeded by the filename

and line number where the error occurs. If more than 10 errors are

encountered, MCP will terminate.

The following error messages are reported by MCP:

6.3.1 Cannot open include file

A '#include' statement on the indicated line specified a

file which could not be opened for reading.

6.3.2 Invalid constant in expression

An expression on the indicated line does not contain a valid

numeric constant at a point where one was expected.

6.3.3 Invalid include file name

A '#include' statement on the indicated line specified a

file name which was not contained within "" or '<>'

characters.

6.3.4 Invalid macro name

A '#define' statement on the indicated line contains a macro

name which does not follow the name rules.

6.3.5 Invalid macro parameter

A '#define' statement on the indicated line contains a macro

parameter name which does not follow the name rules.

A reference to a macro does not have a proper ')' character

to terminate the parameter list.

6.3.6 Invalid operator in expression

An expression on the indicated line does not contain a

supported operator (See #if) at a point where one was expected.

6.3.7 Too many errors

More than 10 errors has been encountered and MCP is

terminating.

6.3.8 Too many macro definitions

MCP has encountered more '#define' statements than it can

handle.

6.3.9 Too many macro parameters

A '#define' statement on the indicated line specifies more

parameters to the macro than MCP can handle.

6.3.10 Too many include files

MCP has encountered more nested '#include' statements than

it can handle.

6.3.11 Undefined macro

A '#undef' or '#forget' statement on the indicated line

references a macro name which has not been defined.

6.3.12 Unterminated comment

The END OF FILE has been encountered while processing a

comment.

6.3.13 Unterminated string

A quoted string on the indicated line has no end. To

continue a string to the next line, use '\' as the last

character on the line. The '\' will not be included in the

string.

7. THE MICRO-C COMPILER

The heart of the MICRO-C programming environment is the COMPILER.

This program reads a file containing a 'C' source program, and

translates it into an equivalent assembly language program.

The compiler includes its own limited pre-processor, which is

suitable for compiling programs requiring only non-parameterized

MACRO substitution, simple INCLUDE file capability, and single-level

CONDITIONAL processing.

7.1 The MCC51 command

The format of the MICRO-C 8051 Compiler command line is:

MCC51 [input_file] [output_file] [options]

[input_file] is the name of the source file containing 'C'

statements to read. If no filenames are given, MCC51 will read

from standard input.

[output_file] is the name of the file to which the generated

assembly language code is written. If less than two filenames are

specified, MCC51 will write to standard output.

7.1.1 Command Line Options

-c - Includes the 'C' source code in the output file

as assembly language comments.

-f - Causes the compiler to "Fold" its literal pool.

(Identical strings not contained in explicit

variables only once in memory).

-l - Enables MCC51 to accept line numbers.

(At beginning of line, followed by ':').

-q - Instructs MCC51 to be quiet, and not display the

startup message when it is executed.

8051: m=0-4 - Specify the memory model to use:

(0=Tiny, 1=Small, 2=Compact, 3=Medium, 4=Large)

8. THE MICRO-C OPTIMIZER

The MICRO-C optimizer is an output code filter which examines the

assembly code produced by the compiler, recognizing known patterns of

inefficent code (using the "peephole" technique), and replaces them with more optimal code which performs the same function. It is entirely table driven, allowing it to be modified for virtually any processor.

Due its many table lookup operations, the optimizer may perform quite slowly when processing a large file. For this reason, most people prefer not to optimize during the debugging of a program, and utilize the optimizer only when creating the final copy.

8.1 The MCO51 command

The format of the MICRO-C Optimizer command line is:

```
MCO51 [input_file] [output_file] [options]
```

[input_file] is the name of the source file containing assembly statements to read. If no filenames are given, MCO51 will read from standard input.

[output_file] is the name of the file to which the optimized assembly code is written. If less than two filenames are specified, MCO51 will write to standard output.

8.1.1 Command Line Options

MCO51 accepts the following command line [options]:

-d - Instructs MCO51 to produce a 'debug' display on standard output showing the source code which it is removing and replacing in the input file.

NOTE: If you do not specify an explict output file, you will get the debug statements intermixed with the optimized code on standard output.

-q - Instructs MCO51 to be quiet, and not display the startup message when it is executed.

9. THE SOURCE LINKER

Many small development environments have assemblers which do not directly support an object linker. This causes a problem with 'C' development, because the library functions must be included in the source code, with several drawbacks:

1) There is no way to automatically tell which library functions to include, therefore, you must do it manually.

2) 'C' library functions must be re-compiled every time, in order to avoid conflict between compiler generated labels.

The MICRO-C Source Linker (SLINK) helps overcome these problems, by automatically joining previously compiled (assembly language) source code from the library to your programs. Only those files containing functions which you reference are joined (Taking into consideration functions called by the included library functions etc...). As the files are joined, compiler generated labels are adjusted to be unique within each file.

9.1 The SLINK Command

The format of the SLINK command line is:

```
SLINK input_file... output_file [options]
```

"input_file" is the name of the source file containing the compiler output from your program. Several input files may be specified, in which case they will all be processed into a single output file.

"output_file" is the name of the file to which the linked source code is written.

9.1.1 Command line options

SLINK accepts the following command line [options]:

```
? - Display command line help summary.
-c - Removes all comments from the output file
c=char - Identifies the character identifying comment
lines. Default is '*'.
f=file - Specify a file containing source file names
i=name - Specify name of the External Index File.
-l - Instructs SLINK to list each library used.
l=path - Identifies the directory path which will be
taken to reference "library" files. If not
specified, it defaults to: '\mc\slib'
p=char - Identifies the PREFIX character of compiler
generated symbols. Defaults is '?'.
-q - Inhibit display of the startup message.
s=file - specify a file to override (replace) the first
"prefix" file defined in the index.
NOTE: File must be in library directory.
t=string- Prefix to prepend to temporary filenames.
If not specified, default is "$".
-w - Sort WORD data to beginning of allocated bulk
unitialized storage (See $DD: directive).
```

9.2 Multiple Input Files

SLINK accepts multiple input files on the command line, and processes each file to build the output file. This allows you to "link" together several previously compiled (or assembly language) modules into one final program.

Any external references which are not resolvable from the

library are assumed to be resolved by one of the input source
files. Since SLINK does not have knowledge of the public symbols
defined in the input files, the absence of an externally
referenced symbol will not be detected until the assembly step
(where it will cause an undefined symbol error).

If you have too many input files to specify on the command line
(128 characters), you can use the F=file option to read the
filenames from a file (one per line). These filenames are
interpreted as if they had all been entered on the command line at
the point where the F= option occurs.

9.3 The External Index File

SLINK uses a special file from the library to determine which
symbols are in which files. This files is called the EXTERNAL
INDEX FILE, and is found in the library directory (see 'l='
option), under the name "EXTINDEX.LIB".

This file contains entries which cross-reference external
symbols to files. Each entry is as follows:

1) Any lines beginning with '<' contain the names of files
which are to be processed and included at the BEGINNING
of the program (Before your source file). This the best
way to include the startup code and any runtime library
routines which are required at all times, and also
provides a method of initializing any segments used.

eg: <6809rl.asm

Note: You can specify up to 5 of these "prefix" files,
and the "S=" command line option allows you to override

the first prefix file defined with any other file in

the library directory. This allows you to place default

system/memory map information (EQU, ORG etc.) in the

first file, and substitute alternate information files

for different system configurations.

2) Any lines beginning with '^' contain the names of files

which are to be processed AFTER the program and library

source files, but BEFORE any uninitialized data areas

are output. This is the best way to set up the location

and storage class of the uninitialized data if it does

not immediately follow the executable program code, and

also providing any postamble needed by the segments.

3) Any lines beginning with '>' contain the names of files

which are to be processed at the END of the program,

after all other information is output. This is the best

way to define heap memory storage, and to provide any

post-amble needed by the assembler.

4) Any lines beginning with '-' contain the names of files

which are to be included if any of the following

symbols (Up to another '<', '^', '>', '-' or '$') are

referenced.

eg: -printf.asm format.asm fgets.asm fget.asm

NOTE: In most cases, the library functions will contain

indications of any external references that they do, in

which case SLINK will automatically include those files

even of the names are not mentioned on the '-' line. In

the example above, the following would suffice:

-printf.asm

5) The names of each symbol which may be referenced

externally must follow the '<', '^', '>' or '-' entry.

Symbols must occur one per line, with no leading or

trailing spaces.

eg: printf

fprintf

sprintf

6) A line beginning with '$' is used to define the pseudo-

opcode used by SLINK to reserve uninitialized data at

the end of the output file. Only one line beginning

with '$' should be entered into the EXTINDEX.LIB file.

The remainder of this line, including all spaces etc.

is entered between each symbol name, and the decimal

size (in bytes) which is written to the output file.

eg: '$ RMB ' <- Quotes are for clarity

A complete example:

-printf.asm

printf

fprintf

sprintf

-scanf.asm

scanf

fscanf

sscanf

<PREFIX.asm

^MIDDLE.ASM

>SUFFIX.ASM

$ RMB

In summary, the output file is written from:

1 - The '<' (prefix) files *

2 - The program source files *\

3 - Library files referenced (if any) * > See note

4 - The '^' (middle) files */

5 - Segments 1-9 from above files *

6 - Uninitialized data definitions (if any)

7 - The '>' (suffix) files

8 - Segments 1-9 from suffix file(s) * See note

* NOTE: If these files contain multiple segments (see later),

all segments are grouped and written in seguential

order. IE: Seg 0 from all files is written, followed

by Seg 1, etc.

9.4 Source file information

9.4.1 SLINK Directives

SLINK interpretes several "directives" which may be inserted

in the input source files to control the source linking

process. These directives must be on a separate line, beginning

in column 1, and must be in uppercase. They are removed by

SLINK during processing, and thus will not cause conflict with

the normal syntax used by the assembler.

$SE:<0-9>

The '$SE' directive is used by SLINK to define multiple

output segments. Up to 10 segments are allowed, with segment 0

being the default which is selected when a file is first
encountered. Other segments (1-9) when selected via this
directive are written to temporary files, and re-joined at the
end of processing in sequential order. this allows you to
separate sections of the source file (such as initialized data,
literal pool etc.) into distinct areas of memory.

$FS:

The '$FS' directive "flushes" the segments, insuring that
all data pending in segments 1-9 are written to the output file
at this point. It is used to terminate (and restart) the
segmentation process from a specific point. This directive also
resets output to segment 0.

$RS:<bit flag number>

The '$RS' directive defines a "runtime library section",
which will only be included if this function is flagged as
being used (See '$RL' directive). To determine if the section
is required, the operands to all '$RL' directives are ORed
together, and then ANDed with the operand to the '$RS' section.
A non-zero result indicates that this section should be
included. '$RS:0' is a special case, and resumes unconditional
output.

$RL:<bit flag number>

The '$RL' directive flags specific functions from the
"runtime library" as being required. Any subsequent '$RS'
sections which have at least one bit in common with any '$RL'
directive will be included.

$DD:<symbol> <size>

The '$DD' directive is used to define uninitialized data storage areas, which will be allocated by SLINK between the '^' (middle) and '>' (suffix) files. This allows you to allocate unitialized data outside of the bounds of the executable image, and thus exclude it from being saved to disk. This action may be thought of as an additional (11'th) segment which is available for uninitialized data only, and which avoids the temporary file read/write overhead associated with use of the other segments.

$EX:<symbol>

The '$EX' directive is used by SLINK to identify any symbols which are externally referenced. Whenever a '$EX' directive is found, SLINK searches the EXTINDX.LIB file for the named symbol, and marks the corresponding files for inclusion in the program.

If you are writing assembly language programs for the library, be sure to include "$SE:<0-9>" directives for any segments you wish to access, "$DD:<symbol> <size>" directives for any uninitialized data you wish to allocate, and "$EX:<symbol>" directives for any symbols which you externally reference. If you wish to place an assembly language comment on the same line, make sure it is separated from the remainder of the directive by at least one space or tab character.

Notes:

- $EX directives will have no effect in MIDDLE or SUFFIX files, because the external references have already been resolved by the time they are processed.

- $DD directives will have no effect in SUFFIX files, because

unitialized storage has already been allocated at that point.

- $RL directives will have no effect on $RS sections which

occur BEFORE them. Normally, $RL is placed in the PREFIX and

PROGRAM files, and $RS is placed in the MIDDLE and SUFFIX

files.

9.4.2 Compiler generated labels

As it processes each source file, SLINK scans each line for

symbols which consist of the '?' character (See 'p=' option),

followed by a number. If it finds such as symbol, it inserts a

two character sequence ranging from 'AA' to 'ZZ' between the

'?', and the number. This sequence will be incremented for each

source file processed, and thus insures that the compiler

generated symbols will be unique for each file.

If you are writing assembly language programs for the

library, you must be careful to avoid using identical local

symbols in any of the library files, one way to do this is to

use symbols which meet the above criteria.

9.5 The SCONVERT command

SCONVERT is a utility which assists in converting existing

assembly language source files into a format which is more

suitable for use by the SLINK. Two main functions are performed:

1) All comments are removed, and all spacing is reduced to a

single space. This minimizes the size of the file, and helps

decrease linkage time.

2) All symbols defined in the file which are not identified as "keep" symbols are converted to resemble the MICRO-C compiler generated symbols. This allows SLINK to adjust them to be unique within each source file.

The format of the SCONVERT command line is:

```
SCONVERT [input_file] [output_file] [options]
```

[input_file] is the name of the source file containing the original assembly language program. If no filenames is given, SCONVERT will read from standard input.

[output_file] is the name of the file to which the converted source code is written. If less that two filenames are given, SCONVERT will write to standard output.

9.5.1 Command line options

SCONVERT accepts the following command line [options]:

? - Display command line help summary.

c=char - Identifies the character used to begin a comment at the trailing end of a source line. If no 'c=' is defined, SCONVERT will terminate processing at the first blank or tab which follows the operand field.

C=char - Identifies the charcter which indicates a comment line in the source code. Defaults to '*'.

k=name - Identifies a symbol name to KEEP. This symbol will not be converted. Multiple 'k=' are permitted.

K=file - Identifies a file containing the names of symbols to KEEP, one per line. Multiple 'K=' are permitted.

p=char - Identifies the PREFIX character which is to be used

for the converted symbols. Defaults to '?'.

-q - Instructs SCONVERT to be quiet, and not issue its

startup message.

SCONVERT identifies symbols in the input source file as any

string beginning with 'A-Z', 'a-z', '_' or '?', and containing

these charcters plus the digits '0-9'. If your assembler source

files uses any other characters in its symbols, you must edit

your sources and change the symbols.

9.6 The SRENUM command

SRENUM is a small utility which re-numbers the compiler

generated symbols within a assembly language source file. This is

useful if you have made added symbols to the file by hand, and

wish to make it "pretty" before adding it to the library etc.

The format of the SRENUM command is:

SRENUM [input_file] [output_file] [options]

9.6.1 Command line options

SRENUM accepts the following command line [options]:

? - Display command line help summary.

p=char - Identifies the PREFIX character which is to be used

to recognize compiler generated symbols.

-q - Instructs SRENUM to be quiet, and not issue its

startup message.

9.7 The SINDEX command

SINDEX is a utility which assists in the creation of the

EXTINDEX.LIB file used by SLINK. When you run SINDEX, it examines all of the '.ASM' files in the current directory, and writes a EXTINDEX.LIB file which contains a '-' type entry for each file, and external symbol entries for any labels which it finds conforming to the 'C' naming conventions (Starts with 'a-z', 'A-Z' or '_', and contains only 'a-z', 'A-Z', '0-9' or '_').

Once you have run SINDEX, you must manually edit the EXTINDEX.LIB file, and remove any file or symbol entries which you do not wish to have available as external references, as well as insert any necessary entries for '<', '^', '>' and '$' commands.

9.7.1 Command line options

SINDEX accepts the following command line options:

? - Display command line help summary.

i=name - Specify name for index file to be written.

Dafault is "EXTINDEX.LIB".

You may also instruct SINDEX to search for a file pattern other than '*.ASM' by passing it as a command line parameter.

eg: SINDEX *.AS1

9.8 The SLIB command

Once you have constructed your source library, you may from time to time want to make minor changes to it, either adding new functions, or removing old ones ones.

You could make such changes simply by editing the EXTINDEX.LIB file, however you would have to be very careful not to add or delete the wrong entry, and you would have to manually determine if adding or removing the file would adversly affect the remainder of the library. For example, you could accidently add a duplicate of another symbol name, or remove a symbol which is referred to by another file.

To simplify maintenance of source libraries, you can make use

of the "Source Librarian", a utility program which automates the

addition and removal of source files, and automatically reports of

any inconsistancies occuring in the source library.

To use SLIB, you must first position yourself to the directory

containing the source library. And then execute SLIB using the

command options described later to indicate the action to be taken

on the library. If you do not specify any actions, SLIB will

simply examine the library and report its size and content.

You may use multiple command options in a single SLIB command

if you wish to add and/or remove more than one file at a time.

After executing any command, SLIB will report on any

inconsistancies which it finds in the library. If any are found,

and you have used a command which caused changes, SLIB will prompt

for permission before writing the updated library file.

9.8.1 Command line options

The following options control the action(s) which will be

performed by SLIB on the source library index file.

? - Display command line help summary.

?=file - Display information about named source file.

a=file - Add specified file as standard functions.

i=index - Use the specified INDEX file, if not specified,

the filename 'EXTINDEX.LIB' is assumed.

m=file - Add named source file as a MIDDLE file.

p=file - Add named source file as a PREFIX file.

-q - Quiet mode: SLIB will not display informational

messages or ask for permission to update index.

r=file - Remove the named file from the index.

s=file - Add named source file as a SUFFIX file.

-w - Write inhibit: SLIB will not write the updated

library index. This is useful if you just want

to see what would happen if you add or remove a

file.

9.9 Making a source library

To make a complete source linkable library, follow these basic

steps:

1) If you have assembly language library functions, run the SINDEX

utility to create an index file with the names of any symbols

defined in them. Edit this file and remove all names which are

LOCAL to the files. Only the global function and variable names

should remain. NOTE: Also leave in any other symbols or names

which you don't want changed by SCONVERT.

2) Use SCONVERT to convert the assembly language sources into

library format, using the index file created above as your KEEP

file (K=EXTINDEX.LIB). You may send the output files directly

to your source library directory.

3) Edit the converted assembly language sources to change any

declarations for uninitialized data into '$DD' directives, and

to add any '$SE' and '$EX' directives which may be needed.

4) Compile all of your 'C' library functions to assembly language,

using a code generator which outputs the appropriate directives for SLINK. Send the ASM output files to your source library

directory.

5) From within your library directory, run the SINDEX utility

again. This will create an index file (EXTINDEX.LIB) which

contains the names of all global symbols.

6) Edit the index file and remove any non-public symbol names. You

should also change the headers for the '<', '^' and '>' files,

and add the '$' record for reserved memory information.

7) Run the SLIB utility, to check the new library for duplicate

symbols, unresolved external references and any other

inconsistancies.

10. THE MAKE UTILITY

The MAKE utility provides a method of automating the building of

larger programs consisting of more that one module. The main benefit

of MAKE is that it keeps track of the files that each module is

dependant on, and will rebuild a module if any of those files have

been modified since the module was last built. This frees the

programmer from the task of remembering which files have been

changed, and the commands needed to rebuild the dependant modules.

10.1 MAKEfiles

To use MAKE, you must first create a MAKEFILE, which is a text

file containing entries for each module in the program. Each entry

consists of a DEPENDANCY list, and a series of COMMANDS.

10.1.1 MAKEfile Entries

A dependency list in MAKE is a line which contains the name

of the module, followed by a ':', followed by the names of any files on which it depends. The module name MUST begin in column 1.

When MAKE is invoked, it will process each dependancy list, and will execute any following commands (up to another dependancy list) if (1) the module does not exist, or (2) if any of the files to the right of the ':' have a timestamp which is later than that of the module. For example:

```
main.asm : main.c main.h \\MC\\stdio.h
\\MC\\mcc51 main.c main.tmp
\\MC\\mco51 main.tmp main.asm
-del main.tmp
```

In the above example, the 'main.asm' would be rebuilt (by compiling and optimizing 'main.c') if either it did not already exist, or any of 'main.c', 'main.h' or '\MC\stdio.h' was found to have a later timestamp.

The '-' preceeding the 'del' command prevents it from being displayed. Unless the '-q' option is enabled, MAKE will display any commands not preceeded by '-' as they are executed.

NOTE: To enter a single '\' in the MAKEFILE, you must use '\\', this is because like 'C', MAKE uses '\' to "protect" special characters which otherwise are used for special functions (such as '\', '$' and '#'). The first '\' "protects" the second one, allowing it to pass through as source text.

10.1.2 Macro Substitutions

Sometimes in a MAKEFILE, you have a single file or directory path that you use over and over again. If it is a long directory path, this may involve a lot of typing, and it

becomes inconvenient to change that name (if you want to use a

different directory etc.) because it is repeated many times.

MAKE includes a MACRO facility, which allows you to define

variable names which will be replaced with a text string when

used in subsequent MAKEfile lines. Names are defined by placing

them in the MAKEfile, followed by '=', and the text string.

Macro names being defined MUST begin in column one, and may

consist of the characters ('a'-'z', 'A'-'Z', '0'-'9', and '_').

Whenever MAKE encounters a '$' in the file, it takes the

name immediately following, and performs the macro replacement:

```
mcdir = \\MC
main.asm : main.c main.h $mcdir\\stdio.h
$mcdir\\mcc51 main.c main.tmp
$mcdir\\mco51 main.tmp main.asm
del main.tmp
```

When a macro name is immediately followed by alphanumeric

text, use a single '\' to separate it from the text. This

"protects" the first character of the text from being

interpreted as part of the macro name:

```
mcdir = \\MC\\
main.asm : main.c main.h $mcdir\stdio.h
$mcdir\mcc51 main.c main.tmp
$mcdir\mco51 main.tmp main.asm
del main.tmp
```

There are several predefined macro symbols which are

available:

$* = The full name of the dependant module (name.type).

$@ = The name only of the dependant module.

$. = The full name of each file in the dependancy list,

separated from each other by a single space.

$, = The full name of each file in the dependancy list,

separated from each other by a single comma.

$: = The name only of each file in the dependancy list,

separated from each other by a single space.

$; = The name only of each file in the dependancy list,

separated from each other by a single comma.

File names in the dependancy list which are preceeded by '-'

will not be included in the '$. $, $: $;' macro expansions:

```
mcdir = \\MC
main.asm : main.c -main.h -$mcdir\\stdio.h
$mcdir\\mcc51 $. $@.TMP
$mcdir\\mco51 $@.TMP $*
del $@.TMP
```

10.1.3 MAKEfile Comments

Whenever MAKE encounters the '#' character in the MAKEFILE,

it treats the remainder of the line as a comment, and does not

process it:

```
# Define Directories
mcdir = \\MC
# Build the MAIN module
main.asm : main.c -main.h -$mcdir\\stdio.h # Dependants
```

```
$mcdir\\mcc51 $. $@.TMP # Compile
$mcdir\\mco51 $@.TMP $* # Optimize
del $@.TMP # Delete tmp
```

10.1.4 Ordering the MAKEfile

MAKE processes the MAKEfile is sequential fashion, with the entries near the top being processed before the entries near the bottom. To insure that each module is built properly, any files appearing in the dependancy list for a module which are themselves dependant on other files, should have MAKEfile entries which occur BEFORE the entries for the modules which are dependant on them:

```
# Define Directories
mcdir = \\MC
# Build the MAIN module
main.asm : main.c -main.h -$mcdir\\stdio.h
$mcdir\\mcc51 $. $@.TMP
$mcdir\\mco51 $@.TMP $*
del $@.TMP
# Build the SUB module
sub.asm : sub.c -sub.h
$mcdir\\mcc51 $. $@.TMP
$mcdir\\mco51 $@.TMP $*
del $@.TMP
# Link the final file & generate listing
# NOTE: If either of the above modules is rebuilt,
# this entry will be guarenteed to execute.
prog.hex : main.asm sub.asm
$mcdir\\slink $. $@.asm l=$mcdir\\lib51
```

```
$mcdir\\asm51 $@ -fs
```

NOTE: For compilers supporting multiple memory models, add the

appropriate m= options and l= options to MCC51 and SLINK

10.2 Directives

Like 'C', MAKE recognizes several "directives" in the MAKEfile.

These directives are only recognized if they occur at the

beginning of the input line:

10.2.1 @include <filename>

This command causes the indicated file to be opened and read

in as the source text. When the end of the new file is

encountered, processing will continue with the line following

"@include" in the original MAKEfile.

10.2.2 @ifdef <name> [name...]

Processes the following lines (up to @else of @endif) only

if one of the given MACRO names is defined. NOTE: <name> should

not be preceeded by '$', otherwise its CONTENTS will be tested.

10.2.3 @ifndef <name> [name...]

Processes the following lines (up to @else of @endif) only

if one of the given MACRO name is NOT defined.

10.2.4 @ifeq <word1> <word2> [word3...]

Processes the following lines (up to @else of @endif) only

if the first word matches one of the remaining words exactly.

This is useful for testing the value of a defined MACRO symbol.

10.2.5 @ifne <word1> <word2>

Processes the following lines (up to @else or @endif) only

if the first word does not match any of the following words.

10.2.6 @else

Processes the following lines (up to @endif) only if the

preceeding @ifdef, @ifndef, @ifeq or @ifne was false.

10.2.7 @endif

Terminates @ifdef, @ifndef, @ifeq and @ifne.

10.2.8 @type <text>

Displays the following text.

10.2.9 @abort [text]

Terminates MAKE with an 'Aborted!' message. Any text on the

remainder of the line will be appended to the message.

10.3 The MAKE command

The format of the MAKE command line is:

MAKE [makefile] [options]

[makefile] is the name of the MAKEfile to process. If no name

is given, MAKE assumes the default name 'MAKEFILE'.

10.3.1 Command Line Options

MAKE accepts the following command line [options]:

? - Causes MAKE to output a short summary of the

available command line options.

-d - Instructs MAKE to operate in "debug" mode, and

display the commands which it would execute,

without actually executing them. This provides

a method of quickly testing the MAKEFILE.

-q - Instructs MAKE to be quiet, and not display the

informational messages and commands executed as

it progresses.

<name>=<text> - Pre-defines a macro of the specified <name>

with the string value <text>. This OVERRIDES

any definition within the MAKEfile, which may

be used to establish a "default" value.

10.4 The TOUCH command

TOUCH is a small utility program which sets the timestamp of

one or more files to the current or specified time/date. It is

useful as a method of forcing MAKE to recognize a file as

"changed", even when it has not.

For example, if you had decided to "undo" several recent

changes by restoring a backup of 'main.c', the restored file will

probably have a timestamp which is older than the last module

which was built. In this case, MAKE would be unaware that the file

has changed, and would therefore not rebuild the module.

The TOUCH command could then be used to "update" the timestamp

of 'main.c' to the current time, causing MAKE to recognize it as a

changed file.

```
TOUCH main.c
```

You could also use TOUCH to force rebuilding of several files:

```
TOUCH main.c sub1.c sub2.c
```

Or even ALL '.C' files:

```
TOUCH *.c
```

TOUCH can also be used to set the timestamp of a file to an

arbitrary value, this may be useful to PREVENT a change from

causing an update:

```
TOUCH main.c t=0:00 d=31/10/80
```

NOTE: Use of the 't=' or 'd=' parameters to TOUCH allows the

possibility that a changed file will go unnoticed. CAUTION is

advised.

The MSDOS implementation of TOUCH supports '-h' and '-s'

options, which cause it to set the timestamp of HIDDEN and/or

SYSTEM files. If these options are not used, TOUCH will not affect

those types of files.

```
+--------------------------------+
| |
| **************************** |
| * The MICRO-C 8051 Library * |
| **************************** |
| |
+--------------------------------+
```

11. The MICRO-C 8051 Library

The library functions described on the following pages are currently available in the MICRO-C library as "standard" 8051

functions.

The prototype for each function listed is preceeded by a MODEL

identifier, which indicates the memory model(s) in which that

particular function (or version of a functions) may be used.

A: - ALL models (TINY, SMALL, COMPACT, MEDIUM and LARGE).

S: - Available in SMALL, COMPACT, MEDIUM and LARGE only.

T: - Available in TINY, SMALL and MEDIUM only.

M: - Available in SMALL and MEDIUM only.

L: - Available in COMPACT and LARGE only

ABS ABS

MODEL & PROTOTYPE:

A: int abs(int number)

ARGUMENTS:

number - Any integer value

RETURN VALUE:

The absolute value of "number"

DESCRIPTION:

The "abs" function returns the absolute value of the argument. If

"number" is a positive value, it is returned unchanged. If negative,

the negate of that value is returned (giving a positive result).

EXAMPLES:

difference = abs(value1 - value2);

ATOI ATOI

MODEL & PROTOTYPE:

S: int atoi(char *string)

A: int _atoi(register char *string)

ARGUMENTS:

string - Pointer to a string containing a decimal number

RETURN VALUE:

16 bit integer value

DESCRIPTION:

The "atoi" function converts an ASCII string in EXTERNAL memory

which contains a signed decimal number (-32768 to 32767) to a 16 bit

value which is returned. An unsigned number of the range (0 to 65535)

may also be used, and the result if assigned to an "unsigned"

variable will be correct.

The "_atoi" function should be used if the string is located in

INTERNAL memory.

EXAMPLES:

value = atoi("1234");

value = atoi("-1");

CHKCH CHKCH

MODEL & PROTOTYPE:

A: int chkch()

ARGUMENTS:

None

RETURN VALUE:

0 - No character available from serial port.

!0 - Character read from serial port.

DESCRIPTION:

The "chkch" function checks to see if a character is ready to be

received from the console serial port. If no character is available,

a zero (0) is returned, otherwise the character is read and its value

is passed back. The CARRIAGE RETURN character (0x0D) will be

translated into a NEWLINE (0x0A) for compatibility with 'C'.

EXAMPLES:

```
if(chkch() == 0x1B) /* Escape KEY pressed ? */
return;
```

CHKCHR CHKCHR

MODEL & PROTOTYPE:

A: int chkchr()

ARGUMENTS:

None

RETURN VALUE:

0-255 - Character read from serial port.

-1 - No character was available.

DESCRIPTION:

The "chkch" function checks to see if a character is ready to be

received from the console serial port. If no character is available,

a -1 is returned, otherwise the character is read and its "raw" value

is passed back (without translation).

EXAMPLES:

if(chkchr() == '\r') /* Return KEY pressed ? */

return;

CONCAT CONCAT

MODEL & PROTOTYPE:

S: register concat(char *dest, char *source, ...)

ARGUMENTS:

dest - Pointer to destination string

source - Pointer to source string

... - Additional sources may be given

RETURN VALUE:

None

DESCRIPTION:

The "concat" function concatinates the given source strings into one destination string. The destination string must be large enough to hold all of the source strings plus the string terminator (zero) byte. No value is returned.

NOTE-1: All strings must be located in EXTERNAL memory.

NOTE-2: This function uses a variable number of arguments, and must be declared as "register" (See "8051io.h").

EXAMPLES:

concat(buffer,"(", string, ")"); /* Enclose string in braces */

DISABLE DISABLE

MODEL & PROTOTYPE:

A: disable()

ARGUMENTS:

None

RETURN VALUE:

None

DESCRIPTION:

The "disable" function disables the 8051 interrupt system, preventing the processor from servicing interrupts. It is used whenever the execution of an interrupt handler may interfere with a particular operation.

When this function is used, the "enable" function should be called

as soon as possible after "disable". Failure to do this may result in

loss of system functions performed under interrupts.

EXAMPLES:

```
disable(); /* Disallow interrupts */
/* ... */
enable(); /* Re-allow interrupts */
```

DELAY DELAY

MODEL & PROTOTYPE:

```
A: delay(unsigned msec)
```

ARGUMENTS:

msec - The number of milliseconds to wait

RETURN VALUE:

None

DESCRIPTION:

This function waits for the specified amount of time, and then

returns. It is used to insert delays in the execution of your

program.

EXAMPLES:

```
P1 = 0x01; /* Raise signal */
```

delay(100); /* Wait 100 ms */

P1 = 0x00; /* Lower signal */

ENABLE ENABLE

MODEL & PROTOTYPE:

A: enable()

ARGUMENTS:

None

RETURN VALUE:

None

DESCRIPTION:

The "enable" function enables the 8051 interrupt system, allowing

the processor to service interrupts. It should be called as soon as

possible following the use of the "disable" function.

EXAMPLES:

disable(); /* Disallow interrupts */

/* ... */

enable(); /* Re-allow interrupts */

EXIT EXIT

PROTOTYPE:

exit()

ARGUMENTS:

None

RETURN VALUE:

N/A - Function never returns

DESCRIPTION:

This function terminates the execution of the program, and returns control to the on-board monitor.

EXAMPLES:

```
exit(); /* Return to monitor */
```

FREE FREE

MODEL & PROTOTYPE:

S: free(char *ptr)

ARGUMENTS:

ptr - Pointer to a previously allocated memory block

RETURN VALUE:

None

DESCRIPTION:

The "free" function releases (de-allocates) a block of memory that was obtained via a call to "malloc", and returns it to the heap. This makes it available for use by other memory allocations.

EXAMPLES:

```
if(!(ptr = malloc(BUFSIZ))) /* Allocate a temporary buffer */
return -1; /* Not enough memory */
/* ... use the memory block ... */
free(ptr); /* Release temporary buffer */
```

GETCH GETCH

MODEL & PROTOTYPE:

A: int getch()

ARGUMENTS:

None

RETURN VALUE:

Value of a character read from the serial port.

DESCRIPTION:

This function reads a single character from the console serial port, and returns it as a positive value in the range of 0 to 255. If the character is a carriage return (0x0D), is translated into a NEWLINE (0x0A) for compatibility with 'C'.

EXAMPLES:

```
while(getch() != 0x1B); /* Wait for an ESCAPE character */
```

GETCHR GETCHR

MODEL & PROTOTYPE:

A: int getchr()

ARGUMENTS:

None

RETURN VALUE:

Value of a character read from the serial port.

DESCRIPTION:

This function reads a single character from the console serial port, and returns it as a positive value in the range of 0 to 255. The character is read in "raw" format, with no translations performed.

EXAMPLES:

```
while(getchr() != '\r'); /* Wait for a RETURN */
```

GETSTR GETSTR

MODEL & PROTOTYPE:

S: int getstr(char *buffer, int size)

A: int _getstr(register char *buffer, int size)

ARGUMENTS:

buffer - Pointer to string to receive line size - Maximum size of line to read

RETURN VALUE:

The number of characters received into the line.

DESCRIPTION:

The "getstr" function reads a string of character from the serial

port and places them into the specified character buffer until either

a NEWLINE character is encountered, or the limit of "size" characters

are read.

The string is terminated with the standard NULL (00) character.

The trailing NEWLINE '\n' character is NOT included in the output

buffer.

The BACKSPACE or DEL keys can be used to edit the input line.

The "buffer" must be located in EXTERNAL memory when the "getstr"

function is used, and INTERNAL memory when the "_getstr" function is

used.

EXAMPLES:

getstr(input_line, 80);

ISALNUM ISALNUM

MODEL & PROTOTYPE:

A: int isalnum(char c)

ARGUMENTS:

c - Any character value

RETURN VALUE:

1 if 'c' is alphabetic or numeric

0 if 'c' is not alphabetic or numeric

DESCRIPTION:

Returns TRUE (1) if the passed character 'c' is an ASCII

alphabetic letter in either upper or lower case or if 'c' is a

numeric digit, otherwise FALSE (0) is returned.

EXAMPLES:

```
while(isalnum(*ptr)) /* Copy over symbol name */
*name++ = *ptr++;
```

ISALPHA ISALPHA

MODEL & PROTOTYPE:

A: int isalpha(char c)

ARGUMENTS:

c - Any character value

RETURN VALUE:

1 if 'c' is alphabetic

0 if 'c' is not alphabetic

DESCRIPTION:

Returns TRUE (1) if the passed character 'c' is an ASCII alphabetic letter in either upper or lower case, otherwise FALSE (0)

is returned.

EXAMPLES:

```
flag = isalpha(input_char);
```

ISASCII ISASCII

PROTOTYPE:

```
int isascii(int c)
```

ARGUMENTS:

c - Any character value

RETURN VALUE:

1 if 'c' is an ASCII character

0 if 'c' is not an ASCII character

DESCRIPTION:

Returns TRUE (1) if the passed character 'c' is a valid ASCII

character (0x00-0xFF), otherwise FALSE (0) is returned.

EXAMPLES:

```
if(!isascii(*ptr))
abort("Invalid character data");
```

ISCNTRL ISCNTRL

MODEL & PROTOTYPE:

A: int iscntrl(char c)

ARGUMENTS:

c - Any character value

RETURN VALUE:

1 if 'c' is a "control" character

0 if 'c' is not a "control" character

DESCRIPTION:

Returns TRUE (1) is the passed character 'c' is an ASCII "control"

character (0x00-0x1F or 0x7F), otherwise FALSE (0) is returned.

EXAMPLES:

```
putch(iscntrl(c) ? '.' : c); /* Display controls as '.' */
```

ISDIGIT ISDIGIT

MODEL & PROTOTYPE:

A: int isdigit(char c)

ARGUMENTS:

c - Any character value

RETURN VALUE:

1 if 'c' is numeric

0 if 'c' is not numeric

DESCRIPTION:

Returns TRUE (1) is the passed character 'c' is an ASCII digit

('0'-'9'), otherwise FALSE (0) is returned.

EXAMPLES:

value = 0;

while(isdigit(*ptr))

value = (value * 10) + (*ptr++ - '0');

ISGRAPH ISGRAPH

MODEL & PROTOTYPE:

A: int isgraph(char c)

ARGUMENTS:

c - Any character value

RETURN VALUE:

1 if 'c' is a non-space printable character

0 if 'c' is a space or not printable

DESCRIPTION:

Returns TRUE (1) if the passed character 'c' is a printable ASCII

character other than a space character (0x21-0xFE), otherwise FALSE

(0) is returned.

EXAMPLES:

```
putch(isgraph(c) ? c : '.');
```

ISLOWER ISLOWER

MODEL & PROTOTYPE:

A: int islower(char c)

ARGUMENTS:

c - Any character value

RETURN VALUE:

1 if 'c' is lower case alphabetic

0 if 'c' is not lower case alphabetic

DESCRIPTION:

Returns TRUE (1) if the passed character 'c' is an ASCII

alphabetic letter of lower case, otherwise FALSE (0) is returned.

EXAMPLES:

```
flag = islower(input_char);
```

ISPRINT ISPRINT

MODEL & PROTOTYPE:

A: int isprint(char c)

ARGUMENTS:

c - Any character value

RETURN VALUE:

1 if 'c' is a printable character

0 if 'c' is not printable

DESCRIPTION:

Returns TRUE (1) if the passed character 'c' is a printable ASCII

character (0x20-0xFE), otherwise FALSE (0) is returned.

EXAMPLES:

putch(isprint(c) ? c : '.');

ISPUNCT ISPUNCT

MODEL & PROTOTYPE:

A: int ispunct(char c)

ARGUMENTS:

c - Any character value

RETURN VALUE:

1 if 'c' is a printable non-alphanumeric character

0 if 'c' is not printable or alphanumeric

DESCRIPTION:

Returns TRUE (1) if the passed character 'c' is a printable ASCII

character which is not a letter of the alphabet or a numeric digit,

otherwise FALSE (0) is returned.

EXAMPLES:

```
while(ispunct(*ptr))
++ptr;
```

ISSPACE ISSPACE

MODEL & PROTOTYPE:

```
A: int isspace(char c)
```

ARGUMENTS:

c - Any character value

RETURN VALUE:

1 if 'c' is a space character (space, tab or newline)

0 if 'c' is not a space character

DESCRIPTION:

Returns TRUE (1) if the passed character 'c' is one of a space,

tab or newline, otherwise FALSE (0) is returned.

EXAMPLES:

```
while(isspace(*ptr))
++ptr;
```

ISUPPER ISUPPER

MODEL & PROTOTYPE:

A: int isupper(char c)

ARGUMENTS:

c - Any character value

RETURN VALUE:

1 if 'c' is upper case alphabetic

0 if 'c' is not upper case alphabetic

DESCRIPTION:

Returns TRUE (1) if the passed character 'c' is an ASCII

alphabetic letter of upper case, otherwise FALSE (0) is returned.

EXAMPLES:

flag = isupper(input_char);

ISXDIGIT ISXDIGIT

MODEL & PROTOTYPE:

A: int isxdigit(char c)

ARGUMENTS:

c - Any character value

RETURN VALUE:

1 if 'c' is a hexidecimal digit

0 if 'c' is not a hexidecimal digit

DESCRIPTION:

Returns TRUE (1) is the passed character 'c' is an valid ASCII

hexidecimal digit ('0'-'9', 'A'-'F', 'a'-'f'), otherwise FALSE (0) is

returned.

EXAMPLES:

```
value = 0;

while(isxdigit(*ptr))

value = (value * 16) +

(isdigit(*ptr) ? *ptr++ - '0' : toupper(*ptr++) - ('A'-10));
```

LONGJMP LONGJMP

MODEL & PROTOTYPE:

M: longjmp(char savenv[3], int rvalue)

T: _longjmp(register char savenv[3], int rvalue)

L: longjmp(char savenv[5], int rvalue)

L: _longjmp(register char savenv[5], int rvalue)

ARGUMENTS:

savenv - Save area for program context

rvalue - Value to be returned by "setjmp"

RETURN VALUE:

N/A - Function never returns

DESCRIPTION:

The "longjmp" functions causes execution to transfer to the

"setjmp" call which set up the "savenv" variable. The "setjmp" function will appear to return the value of "rvalue".

NOTE-1: "longjmp" may only be used from within the function

calling "setjmp" or a function which has been called "beneath" that

function. IT MUST NOT BE USED AFTER THE FUNCTION CALLING "SETJMP" HAS

TERMINATED.

NOTE-2: If "rvalue" is zero, the function calling "setjmp" will

assume that it is returning from initialization. Unless you want this

unusual behavior, you should not pass a return value of zero to

"longjmp".

See also SETJMP.

EXAMPLES:

```
if(chkch() == ('C'-'@')) /* If Control-C entered... */

longjmp(savearea, 1); /* Return to main function */
```

LONGMATH LONGMATH

PROTOTYPES:

```
int longadd(register char *num1, register char *num2);

int longsub(register char *num1, register char *num2);

longmul(register char *num1, register char *num2);

longdiv(register char *num1, register char *num2);

int longshr(register char *num1);

int longshl(register char *num1);

longcpy(register char *num1, register char *num2);

longset(register char *num1, unsigned value);

int longtst(register char *num1);

int longcmp(register char *num1, register char *num2);
```

extern register char Longreg[];

ARGUMENTS:

num1 - First LONG operand, receives result if generated

num2 - Second LONG operand, is not altered

value - 16 bit value to initialize LONG number

RETURN VALUE:

longadd - 0 = Ok, 1 = addition overflowed

longsub - 0 = Ok, 1 = subtraction underflowed

longshr - Carry out of shift (1/0)

longshl - Carry out of shift (1/0)

longtst - 0 = Number is zero, !0 = Number is not zero

longcmp - 0 = (num1 == num2), 1 = (num1 > num2), -1 = (num1 < num2)

DESCRIPTION:

This set of functions performs basic arithmetic functions on LONG

numbers:

longadd(num1, num2) -> num1 += num2

longsub(num1, num2) -> num1 -= num2

longmul(num1, num2) -> num1 *= num2 , Longreg = num1 * num2

longdiv(num1, num2) -> num1 /= num2 , Longreg = num1 % num2

longshr(num1) -> num1 >>= 1

longshl(num1) -> num1 <<= 1

longcpy(num1, num2) -> num1 = num2

longset(num1, value) -> num1 = (long) value

longtst(num1) -> Test (num1 != 0)

longcmp(num1, num2) -> Compare num1 and num2

As shipped, a LONG number is 32 bits (4 bytes) in size, however this can be changed by altering the LONGMATH.ASM in the library.

For a complete example of using these functions, refer to the

LONGCALC.C example program included with the package.

MALLOC MALLOC

MODEL & PROTOTYPE:

S: char *malloc(int size)

ARGUMENTS:

size - Size of memory block to allocate (in bytes).

RETURN VALUE:

0 - Memory allocation failed

!0 - Pointer to allocated memory block

DESCRIPTION:

The "malloc" function allocates a block of memory of the specified

size from the heap, and returns a pointer to it. This memory will

remain allocated until it is explicitly released with the "free"

function.

NOTE: All memory allocated by "malloc" is EXTERNAL.

EXAMPLES:

```
if(!(ptr = malloc(BUFSIZ))) /* Allocate a temporary buffer */
return -1; /* Not enough memory */
/* ... use the memory block ... */
free(ptr); /* Release temporary buffer */
```

free(ptr); /* Release temporary buffer */

MAX MAX

MODEL & PROTOTYPE:

A: int max(int value1, int value2)

ARGUMENTS:

value1 - Any integer value

value2 - Any integer value

RETURN VALUE:

The greater of "value1" or "value2"

DESCRIPTION:

The "max" function returns the higher of its two argument values.

EXAMPLES:

biggest = max(a, b);

MEMSET MEMSET

MODEL & PROTOTYPE:

S: memset(char *block, char value, unsigned size)

A: _memset(register char *block, char value, unsigned size)

ARGUMENTS:

block - Pointer to a block of memory

value - Value to initialize memory with

size - Number of bytes to initialize

RETURN VALUE:

None

DESCRIPTION:

Sets a block of memory beginning at the pointer "block", for

"size" bytes to the byte value "value".

EXAMPLES:

memset(buffer, 0, 100);

MOVE MOVE

MODEL & PROTOTYPE:

S: move(char *dest, char *source, unsigned size)

S: move_(char *dest, register char *source, unsigned size);

A: _move(register char *dest, char *source, unsigned size);

A: _move_(register char *dest, register char *source, unsigned size);

ARGUMENTS:

dest - Pointer to the destination memory

source - Pointer to the souce memory

size - Number of bytes to copy

RETURN VALUE:

None

DESCRIPTION:

The "move" functions will copy the specified number of bytes from

the source to the destination addresses in memory.

EXAMPLES:

```
move(buffer1, buffer2, 256);
```

MIN MIN

MODEL & PROTOTYPE:

```
A: int min(int value1, int value2)
```

ARGUMENTS:

value1 - Any integer value

value2 - Any integer value

RETURN VALUE:

The smaller of "value1" or "value2"

DESCRIPTION:

The "min" function returns the lower of its two argument values.

EXAMPLES:

```
least = min(a, b);
```

NARGS NARGS

MODEL & PROTOTYPE:

A: int nargs()

ARGUMENTS:

None

RETURN VALUE:

The number of arguments passed to the calling function

DESCRIPTION:

Returns the number of arguments passed to a "register" function.

NOTE: When calling a "register" function, MICRO-C loads the

accumulator with the number of arguments just prior to calling the

function. This "nargs" routine is simply a null definition which

returns with the same value in the accumulator as was there when it

was called. Therefore "nargs" MUST BE THE FIRST ARITHMETIC ENTITY

EVALUATED WITHIN THE REGISTER FUNCTION or the contents of the

accumulator will be lost. Some examples of "register" definitions and

the use of "nargs" may be found in the library source code.

EXAMPLES:

first_arg = nargs() * 2 + &arguments;

PEEK PEEK

MODEL & PROTOTYPE:

A: int peek(unsigned address)

ARGUMENTS:

address - 16 bit external memory address

RETURN VALUE:

The 8 bit value read from the given external memory address.

DESCRIPTION:

The "peek" function reads and returns a byte (8 bits) from

external memory as an integer value betweek 0 and 255.

When using the TINY memory model, "peek" provides a method by

which the program can access the external DATA memory space.

When using the SMALL & MEDIUM memory models, its usually more

efficent to access external memory through a pointer (or cast to

pointer), than to call "peek".

EXAMPLES:

var = peek(0);

PEEKW PEEKW

MODEL & PROTOTYPE:

A: int peekw(unsigned address)

ARGUMENTS:

address - 16 bit external memory address

RETURN VALUE:

The 16 bit value read from the given external memory address.

DESCRIPTION:

The "peekw" function reads and returns a word (16 bits) from

external memory as an integer value betweek 0 and 65535 (-1).

When using the TINY memory model, "peekw" provides a method by

which the program can access the external DATA memory space.

When using the SMALL & MEDIUM memory models, its usually more

efficent to access external memory through a pointer (or cast to

pointer), than to call "peekw".

EXAMPLES:

var = peekw(0);

POKE POKE

MODEL & PROTOTYPE:

A: poke(unsigned address, int value)

ARGUMENTS:

address - 16 bit external memory address

value - Value to be written to memory

RETURN VALUE:

None

DESCRIPTION:

The "poke" function writes a byte (8 bit) value to external

memory.

When using the TINY memory model, "poke" provides a method by

which the program can access the external DATA memory space.

When using the SMALL & MEDIUM memory models, its usually more

efficent to access external memory through a pointer (or cast to

pointer), than to call "poke".

EXAMPLES:

poke(0, 0); /* Write 0 to location 0 */

POKEW POKEW

MODEL & PROTOTYPE:

A: pokew(unsigned address, int value)

ARGUMENTS:

address - 16 bit external memory address

value - Value to be written to memory

RETURN VALUE:

None

DESCRIPTION:

The "pokew" function writes a word (16 bit) value to external

memory.

When using the TINY memory model, "pokew" provides a method by

which the program can access the external DATA memory space.

When using the SMALL & MEDIUM memory models, its usually more

efficent to access external memory through a pointer (or cast to pointer), than to call "pokew".

EXAMPLES:

pokew(0, 0); /* Write 0 to location 0 and 1 */

PRINTF PRINTF

MODEL & PROTOTYPE:

A: register printf(char *format, arg, ...)

ARGUMENTS:

format - Pointer to format string

arg - Argument as determined by format string

... - Additional arguments may be required

RETURN VALUE:

None

DESCRIPTION:

The "printf" function performs a formatted print to console serial

port. The 'format' string is written to the serial port with the

arguments substituted for special "conversion characters". These

"conversion characters" are identified by a preceeding '%', and may

be one of the following:

b - Binary number

c - Character

d - Decimal (signed) number

i - String from INTERNAL memory

o - Octal number

s - String from EXTERNAL memory

u - Unsigned decimal number

x - Hexidecimal number

% - A single percent sign (No argument used)

A numeric "field width" specifier may be placed in between the '%'

and the conversion character, in which case the value will be output

in a field of that width. If the "field width" is a negative number,

the output will be left justified in the field, otherwise it is right

justified. If the field width contains a leading '0', then the output

field will be padded with zero's, otherwise spaces are used.

If no "field width" is given, the output is free format, using

only as much space as required.

NOTE: This function uses a variable number of arguments, and must

be declared as "register" (See "8051io.h").

EXAMPLES:

printf("Hello world!!!\n");

printf("%u interrupts have occured!\n", int_count);

PUTCH PUTCH

MODEL & PROTOTYPE:

A: putch(char c)

ARGUMENTS:

c - Any character value

RETURN VALUE:

None

DESCRIPTION:

This function writes the character 'c' to the console serial port.

The NEWLINE character is translated into a LINE-FEED, followed by a

CARRIAGE return, which causes the printing position to advance to the

first column of the next line.

EXAMPLES:

putch('*');

putch('\n');

PUTCHR PUTCHR

MODEL & PROTOTYPE:

A: putchr(char c)

ARGUMENTS:

c - Any character value

RETURN VALUE:

None

DESCRIPTION:

This function writes the character 'c' to the console serial port.

The character is written in "raw" format, with no translations of any kind.

EXAMPLES:

putchr('*');

putchr('\r');

PUTSTR PUTSTR

MODEL & PROTOTYPE:

A: putstr(char *string)

A: _putstr(register char *string)

ARGUMENTS:

string - Pointer to a character string

RETURN VALUE:

0 if successful, otherwise an operating system error code

DESCRIPTION:

The "putstr" function writes the specified string to the serial

port. The zero terminating the string is NOT written.

EXAMPLES:

putstr("Text message");

RAND RAND

MODEL & PROTOTYPE:

A: unsigned rand(unsigned limit)

ARGUMENTS:

limit - Maximum value to return

RETURN VALUE:

A pseudo-random number in the range of 0 to (limit-1)

DESCRIPTION:

The "rand" function calculates the next value of a pseudo-random sequence, based on a 16 bit unsigned "seed" value, which it maintains in the global variable "RANDSEED". The new value is stored as the new seed value, and is then divided by the "limit" parameter, to obtain the remainder, which is returned. This results in a random number in the range of zero (0) to (limit - 1).

Any particular sequence may be repeated, by reseting the "RANDSEED" value.

EXAMPLES:

```
extern register int RANDSEED;
/* ... */
RANDSEED = 1234; /* Set the random seed value */
value = rand(52); /* Generate a pseudo-random value */
```

SERINIT SERINIT

MODEL & PROTOTYPE:

A: serinit(unsigned speed)

ARGUMENTS:

speed - Desired serial port BAUD RATE

RETURN VALUE:

None

DESCRIPTION:

The "serinit" function initializes the 8051 SERIAL PORT, and TIMER 1 hardware to enable serial communication at the specified baud rate.

If you are using system which does not run at the standard clock rate of 11.0592 Mhz, you must modify the "conversion constant" in the SERINIT.ASM file of the compiler library. This constant is calculated as the CPU CLOCK (in Hz) divided by 384.

For example:

11.0592 Mhz = 11059200 Hz / 384 = 28800

7.3728 Mhz = 7372800 Hz / 384 = 19200

The value written to the timer is computed as:

timer_value = -(CONVERSION_CONSTANT/desired_speed) & 0xFF;

If you are using a non-standard baud rate or crystal frequency, you should check that the actual speed generated, which is calculated as:

speed = CONVERSION_CONSTANT/timer_value;

will be within 5% of the desired speed. If it is not, then that particular speed is not available with your hardware configuration.

Remember to drop any fractional portion of "timer_value" when calculating the actual speed.

EXAMPLES:

serinit(2400); /* Set speed to 2400 BPS */

SETJMP SETJMP

MODEL & PROTOTYPE:

M: int setjmp(char savenv[3])

T: int _setjmp(register char savenv[3]);

L: int setjmp(char savenv[5])

L: int _setjmp(register char savenv[5]);

ARGUMENTS:

savenv - Save area for program context

RETURN VALUE:

0 is returned when actually called

Value passed to "longjmp" is returned otherwise

DESCRIPTION:

When called, the "setjmp" function stores the current execution

state of the program into the passed integer array, and returns the

value zero.

The "longjmp" function may then be used to return the program to

the "setjmp" call. In this case, the value returned by "setjmp" will

be the value which was passed to "longjmp". This allows the function

containing "setjmp" to determine which call to "longjmp" transfered

execution to it.

See also LONGJMP.

EXAMPLES:

```
switch(setjmp(savearea)) {
case 0 : printf("Longjmp has been set up"); break;
case 1 : printf("Control-C Interrupt"); break;
case 2 : printf("Reset command executed"); break;
default: printf("Software error trap"); break; }
```

SPRINTF SPRINTF

MODEL & PROTOTYPE:

S: register sprintf(char *dest, char *format, arg, ...)

ARGUMENTS:

dest - Pointer to destination string

format - Pointer to format string

arg - Argument as determined by format string

... - Additional arguments may be required

RETURN VALUE:

None

DESCRIPTION:

The "sprintf" routine performs a formatted print to a string in memory. The "format" string is written to the destination string with the arguments substituted for special "conversion characters".

See "printf" for more information on format strings.

NOTE: This function uses a variable number of arguments, and must be declared as "register" (See "8051io.h").

EXAMPLES:

```
sprintf(buffer, "Count=%u", count);
```

SQRT SQRT

MODEL & PROTOTYPE:

A: int sqrt(unsigned value)

ARGUMENTS:

value - Number for which to calculate square root

RETURN VALUE:

The integer square root (rounded up) of the argument value

DESCRIPTION:

The SQRT function returns the smallest number which when

multiplied by itself will give a number equal to or larger that the

argument value.

EXAMPLES:

```
/*
* Draw a circle about point (x, y) of radus (r)
*/
circle(x, y, r)
int x, y, r;
{
```

```
int i, j, k, rs, lj;

rs = (lj = r)*r;

for(i=0; i <= r; ++i) {

j = k = sqrt(rs - (i*i));

do {

plot_xy(x+i, y+j);

plot_xy(x+i, y-j);

plot_xy(x-i, y+j);

plot_xy(x-i, y-j); }

while(++j < lj);

lj = k; }

}
```

STRCAT STRCAT

MODEL & PROTOTYPE:

S: char *strcat(char *dest, char *source)

ARGUMENTS:

dest - Pointer to destination string

source - Pointer to source string

RETURN VALUE:

Pointer to zero terminating destination string

DESCRIPTION:

This function concatinates the source string onto the tail of the

destination string. The destination string must be large enough to hold the entire contents of both strings.

The strings must be in EXTERNAL memory.

EXAMPLES:

```
strcat(program_name, ".c");
```

STRCHR STRCHR

MODEL & PROTOTYPE:

```
S: char *strchr(char *string, char chr)
```

ARGUMENTS:

string - Pointer to a character string

chr - Character to look for

RETURN VALUE:

Pointer to the first occurance of 'chr' in 'string'

Zero (0) if character was not found

DESCRIPTION:

Searches the passed string got the first occurance of the

specified character. If the character is found, a pointer to its

position in the string is returned. If the character is not found, a

null pointer is returned.

The null (0) character is treated as valid data by this function,

thus:

```
strchr(string, 0);
```

would return the position of the null terminator of the string.

The strings must be in EXTERNAL memory.

EXAMPLES:

comma = strchr(buffer, ',');

STRCMP STRCMP

MODEL & PROTOTYPE:

S: int strcmp(char *string1, char *string2)

ARGUMENTS:

string1 - Pointer to first string

string2 - Pointer to second string

RETURN VALUE:

0 - Strings match exactly

1 - String1 is greater than string2

-1 - String2 is greater than string1

DESCRIPTION:

This function compares two strings character by character. If the

two strings are identical, a zero (0) is returned. If the first

string is greater than the second (as far as ASCII is concerned), a

one (1) is returned. If the second string is greater, a negative one

(-1) is returned.

The strings must be in EXTERNAL memory.

EXAMPLES:

```
if(!strcmp(command, "quit"))
return;
```

STRCPY STRCPY

MODEL & PROTOTYPE:

S: char *strcpy(char *dest, char *source)

ARGUMENTS:

dest - Pointer to destination string

souce - Pointer to source string

RETURN VALUE:

Pointer to zero terminating destination string

DESCRIPTION:

This function copies the source string to the destination string.

All data is copied up to and including the zero byte which terminates

the string. The destination string must be large enough to hold the

entire source.

The strings must be in EXTERNAL memory.

EXAMPLES:

```
strcpy(buffer, "DEFAULT NAME");
```

STRLEN STRLEN

MODEL & PROTOTYPE:

S: int strlen(char *string)

ARGUMENTS:

string - Pointer to a character string

RETURN VALUE:

The length of the string

DESCRIPTION:

Returns the length in character of the passed string. The length

does not include the zero byte which terminates the string.

The string must be in EXTERNAL memory.

EXAMPLES:

length = strlen(command);

TOLOWER TOLOWER

MODEL & PROTOTYPE:

A: char tolower(char c)

ARGUMENTS:

c - Any character value

RETURN VALUE:

The value of 'c', converted to lower case

DESCRIPTION:

Returns the value of 'c' converted to lower case. If 'c' is not a

letter of upper case, no change is made, and the original value of

'c' is returned.

EXAMPLES:

input_char = tolower(getch());

TOUPPER TOUPPER

MODEL & PROTOTYPE:

A: char toupper(char c)

ARGUMENTS:

c - Any character value

RETURN VALUE:

The value of 'c', converted to upper case

DESCRIPTION:

Returns the value of 'c' converted to upper case. If 'c' is not a

letter of lower case, no change is made, and the original value of

'c' is returned.

EXAMPLES:

putch(toupper(c));

```
+------------------------------+
| |
| ************************** |
| * I2C Library Extensions * |
| ************************** |
| |
+------------------------------+
```

11.1 I2C Library Extensions

The library functions described in this section communicate

with I/O devices over an "I2C" clocked serial interface bus.

The bus protocol is implemented entirely in software, and uses

two of the parallel I/O pins on the 8051/8052 processor. The P1.0

pin is used as the I2C clock line (SCL), and the P3.4/T0 pin is

used as the I2C data line (SDA).

The devices currently supported are:

X2404 or X2416 serial EEPROM.

PCF8591 4 channel A/D, and 1 D/A convertor.

The "BD52" single board computer available from Dunfield

Development Systems contains sockets for these devices, pre-wired to the P1.0 and P3.4/T0 pins as described above.

Additional I2C interface devices can be easily supported, by writing appropriate drivers, and using the low level I2C interface routines, which are contained in the library file I2C.ASM.

The library functions in this section are available in ALL of the 8051 memory models.

ADC ADC

PROTOTYPE:

int adc(unsigned char value);

ARGUMENTS:

value - Output value to DAC

RETURN VALUE:

0 - Successful completion

-1 - Error condition

DESCRIPTION:

This function performs access to a PCF8591 4 channel Analog to Digital convertor (ADC), and 1 channel Digital to Analog convertor (DAC).

When this function is called, it first writes the "value" to the DAC output, and then reads the four inputs. The inputs are configured as "single ended".

The result of the conversions are recorded in the external unsigned character variables ADC0, ADC1, ADC2 and ADC3. Since these variables are in INTERNAL 8051 memory, you must also declare

them as "register". An appropriate declaration is contained in the

header file "8051ADC.H".

Both the output "value", and the four inputs have a range of

0-255, which on the BD52, represent a range of 0-2.5 volts DC.

EXAMPLES:

```
/* Monitor four inputs, and set output to average */
extern register unsigned char ADC0, ADC1, ADC2, ADC3, P1;
main()
{
ADC0=ADC1=ADC2=ADC3=0x80; /* Assume halfway for initial */
for(;;)
adc(((unsigned)ADC0+ADC1+ADC2+ADC3)/4);
}
```

EEREAD EEREAD

PROTOTYPE:

```
int eeread(char *address);
```

ARGUMENTS:

address - Byte address to read in EEPROM

RETURN VALUE:

0-255 - Data read from EEPROM

-1 - Error condition

DESCRIPTION:

This function reads a byte of data from a X2404 or X2416 EEPROM

chip, over the I2C bus. The "address" parameter may range from 0

to 511 for the X2404, and from 0 to 2047 for the X2416.

EXAMPLES:

```
/* Load first 512 byte of external memory from EEPROM */
loadfromee()
{
char *ptr;
for(ptr=0; ptr < 512; ++ptr)
*ptr == eeread(ptr);
}
```

EEWRITE EEWRITE

PROTOTYPE:

```
int eewrite(char *address, char data);
```

ARGUMENTS:

address - Byte address to write in EEPROM

data - Data value to write

RETURN VALUE:

0 - Data written successfully

-1 - Error condition

DESCRIPTION:

This function writes a byte of data to a X2404 or X2416 EEPROM

chip, over the I2C bus. The "address" parameter may range from 0

to 511 for the X2404, and from 0 to 2047 for the X2416.

EXAMPLES:

```
/* Save first 512 byte of external memory from EEPROM */
savetoee()
{
char *ptr;
for(ptr=0; ptr < 512; ++ptr)
eewrite(ptr, *ptr);
}
```

MICRO-C

TABLE OF CONTENTS

MICRO-C Table of Contents

Page