KEIL SOFTWARE

8051 Utilities

BL51 Code Banking Linker/Locator
LIB51 Library Manager
OC51 Banked Object File Converter
OH51 Object Hex Converter

User's Guide 04.95

Every effort was made to ensure accuracy in this manual and to give appropriate credit to persons, companies, and trademarks referenced herein.

A84 D04/27/95

Preface

This manual describes the Keil Software utilities for the 8051. Included are the BL51 code banking linker/locator, the LIB51 library manager, the OC51 banked object file converter, and the OH51 object to hex converter. You use these utilities to generate executable 8051 programs from modules you create using the Keil C51 compiler and A51/A251 assembler and the Intel ASM-51 assembler and PL/M-51 compiler. This user's guide assumes that you are familiar with the MS-DOS operating system and how to program the 8051 microprocessor.

This user's guide is divided into the following five chapters:

"Chapter 1. BL51 Code Banking Linker/Locator," describes the linker and explains how to use the command-line directives. This part includes also reference section of all linker directives, along with examples and descriptions.

"Chapter 2. Application Examples," contains several program examples which show the linker and tool invocation.

"Chapter 3. LIB51 Library Manager," shows you how to use the library manager to create and maintain a library of object modules.

"Chapter 4. OC51 Banked Object File Converter," shows you how to convert banked object files (object files created with the BL51 code banking linker/locator) into absolute object files.

"Chapter 5. OH51 Object-Hex Converter," describes the object file converter program that generates HEX files. This application allows you to create Intel HEX files from the absolute object modules created by the BL51 code banking linker/locator and OC51.

Document Conventions

This document uses the following conventions:

Examples	Description
README.TXT	Bold capital text is used for the names of executable programs, data files, source files, environment variables, and commands you enter at the MS-DOS command prompt. This text usually represents commands that you must type in literally. For example: **CLS** **DIR** **BL51.EXE** Note that you are not required to enter these commands using all capital letters.
`Courier`	Text in this typeface is used to represent information that displays on screen or prints at the printer. This typeface is also used within the text when discussing or describing command line items.
Variables	Text in italics represents information that you must provide. For example, *projectfile* in a syntax string means that you must supply the actual project file name. Occasionally, italics are also used to emphasize words in the text.
Elements that repeat…	Ellipses (…) are used in examples to indicate an item that may be repeated.
Omitted code . . .	Vertical ellipses are used in source code examples to indicate that a fragment of the program is omitted. For example: **`void main (void) {`** **.** **.** **.** **`while (1);`**
⟦*Optional Items*⟧	Optional arguments in command-line and option fields are indicated by double brackets. For example: **`C51 TEST.C PRINT`** ⟦**(*filename*)**⟧
{ *opt1* | *opt2* }	Text contained within braces, separated by a vertical bar represents a group of items from which one must be chosen. The braces enclose all of the choices and the vertical bars separate the choices. One item in the list must be selected.
Keys	Text in this sans serif typeface represents actual keys on the keyboard. For example, “Press **Enter** to continue.”

Contents

Chapter 1. BL51 Code Banking Linker/Locator 1
Introduction to BL51 1
BL51 Overview.. 3
Combining Program Modules 4
Combining Segments 5
Locating Segments 6
Overlaying Data Memory... 8
Resolving External References 8
Absolute Address Calculation.. 8
Generating an Absolute Object File 9
Generating a Listing File.. 9
Bank Switching 10
Using RTX51 and RTX51 Tiny.. 11
Linking Programs with BL51.. 12
Long Command Lines.. 13
Command Files 13
Command-Line Examples.. 13
DOS Errorlevel 14
Output File 14
Command-Line Directives 15
Directive Summary 17
Listing File Directives.. 17
Output File Directives.. 22
Segment Size and Location Directives.. 25
High-Level Language Directives... 32
Bank Switching Directives... 43
RTX 51 Full and RTX51 Tiny Directives.. 50
Bank Switching Configuration... 51
L51_BANK.A51 Constants 51
Public Symbols in L51_BANK.A51 53
Configuration Examples.. 54
BL51 Directive Reference 57
BANKAREA 58
BANK*x* 59
BIT... 60
CODE.. 61
COMMON 62
DATA 63
IDATA... 64
IXREF.. 65
NAME.. 66
NOAMAKE 67
NODEBUGLINES... 68
NODEBUGPUBLICS.. 69

NODEBUGSYMBOLS.....70
NODEFAULTLIBRARY.....71
NOLINES.....72
NOMAP.....73
NOPUBLICS.....74
NOSYMBOLS.....75
OVERLAY / NOOVERLAY.....76
PAGELENGTH.....78
PAGEWIDTH.....79
PDATA.....80
PRECEDE.....81
PRINT.....82
RAMSIZE.....83
REGFILE.....84
RTX51.....85
RTX51TINY.....86
STACK.....87
XDATA.....88
BL51 Error Messages.....89
Warnings.....89
Non-Fatal Errors.....92
Fatal Errors.....95
Exceptions.....98

Chapter 2. Application Examples.....101
C51 Example.....101
CSAMPLE1.C Listing File.....102
CSAMPLE2.C Listing File.....102
CSAMPLE3.C Listing File.....103
CSAMPLE Linker/Locator Listing File.....103
A51 Example.....107
ASAMPLE1.A51 Listing File.....108
ASAMPLE2.A51 Listing File.....109
ASAMPLE3.A51 Listing File.....110
ASAMPLE Linker/Locator Listing File.....111
Code Banking Examples.....112
Example 1. Code Banking with C51.....112
Example 2. Code Banking with Constants.....119
Example 3. Placing Specific Functions in Code Banks.....124
Example 4. Code Banking with PL/M-51.....129

Chapter 3. LIB51 Library Manager.....137
Using LIB51.....137
Interactive Mode.....137
Command Summary.....138
Creating a Library.....139
Adding Object Modules.....140
Removing Object Modules.....141

Listing Library Contents 142
Help Information.. 143
LIB51 Error Messages.. 144
Fatal Errors 144
Errors 145

Chapter 4. OC51 Banked Object File Converter..147
Using OC51 147
OC51 Error Messages.. 149
Fatal Errors 149

Chapter 5. OH51 Object-Hex Converter..151
Using OH51.. 151
OH51 Error Messages.. 152
Intel HEX File Format.. 154
Record Format 154
Data Records.. 155
End-of-File (EOF) Records.. 155
Example Intel HEX File.. 156

Glossary..157

Index..163

Chapter 1. BL51 Code Banking Linker/Locator

Introduction to BL51

The BL51 code banking linker/locator is used to link or join together object modules that were created using the A51 assembler, the C51 compiler, the PL/M-51 compiler. Object modules that are created by these translators are relocatable and cannot be directly executed. They must be converted into absolute object modules. The BL51 code banking linker/locator does this and much more.

NOTE
The BL51 code banking linker/locator provides a superset of the functions performed by the L51 Linker/Locator. BL51 provides support for the following capabilities, which are not available with L51.

- Programs that are larger than 64 KBytes
- Code banking or bank switching
- RTX51 Tiny Real–Time Multitasking Operating System
- RTX51 Full Real–Time Multitasking Operating System

Programs you create using the A51 Assembler and the C51 C Compiler must be linked using the BL51 code banking linker/locator. You cannot execute or simulate programs that are not linked, even if they consist of only one source module. If your application will be using multiple code banks or if you your application will be using either RTX51 or RTX51 Tiny, you must use the BL51 code banking linker/locator to link your program. L51 does not handle the requirements of bank switching or Real–Time applications.

Programs you create using the A51 assembler and the C51 compiler must be linked using the L51 linker/locator or the BL51 code banking linker/locator. If your application will be using multiple code banks, RTX51 Full, or RTX51 Tiny, you must use the BL51 code banking linker/locator to link your program. The L51 linker/locator does not handle the requirements of bank switching or real-time applications.

The BL51 code banking linker/locator will link one or more object modules together and will resolve references from one to the other. This allows you to create a large program that is spread over a number of source and object modules.

The BL51 code banking linker/locator provides the following functions:

- Combines several program modules into one module, automatically incorporating modules from the library files
- Combines relocatable partial segments of the same segment name into a single segment
- Allocates and manipulates the necessary memory for the segments with which all relocatable and absolute segments are processed
- Analyzes the program structure and manipulates the data memory using overlay techniques
- Resolves external and public symbols
- Defines absolute addresses and computes the addresses of relocatable segments
- Produces an absolute object file that contains the entire program
- Produces a listing file that contains information about the Link/Locate procedure, the program symbols, and the cross reference of public and external symbol names
- Detects errors found in the invocation line or during the Link/Locate run.

In addition to the operations performed by the L51 linker/locator, the BL51 code banking linker/locator provides support for the following:

- Programs that are larger than 64 KBytes
- Code banking or bank switching
- RTX51 Tiny Real-Time Multitasking Operating System
- RTX51 Full Real-Time Multitasking Operating System

All of these operations are described in detail in the remaining sections of this chapter.

“BL51 Overview” on page 3 provides you with a summary of the features and capabilities of the BL51 code banking linker/locator. This chapter introduces the concepts of what a linker is and does.

“Linking Programs with BL51” on page 12 describes how to invoke the linker from the command line. The command-line arguments are discussed, and examples are provided.

“Directive Summary” on page 17 lists the command-line directives by category and provides you with descriptions of each, along with command-line examples.

“Bank Switching Configuration” on page 51 describes what bank switching is and how it is implemented by the BL51 code banking linker/locator. This chapter also shows how to make applications that are larger than 64 KBytes work with code banking.

“BL51 Directive Reference” on page 57 provides an alphabetized listing of all of the directives that you can enter on the command line.

“BL51 Error Messages” on page 89 lists the errors that you may encounter when you use the BL51 code banking linker/locator.

BL51 Overview

The BL51 code banking linker/locator takes the object files and library files you specify and generates either an absolute object file or a banked object file. (An absolute object files is generated for a non-code banking program. A banked object file is generated for code banking program.) The BL51 code banking linker/locator also generates a listing or map file.

Absolute object files may be converted into Intel HEX files by the OH51 Object-Hex Converter. Banked object files must be converted by the OC51 Banked Object File Converter into absolute object files (one for each bank) before they can be converted into Intel HEX files by the OH51 Object-Hex Converter.

While processing object and library files, the BL51 code banking linker/locator performs the following operations.

Combining Program Modules

The object modules that the BL51 code banking linker/locator combines are processed in the order in which they are specified on the command line. The BL51 code banking linker/locator processes the contents of object modules created with the A51 assembler or the C51 compiler. Library files, however, contain a number of different object modules; and, only the object modules in the library file that specifically resolve external references are processed by the BL51 code banking linker/locator.

Segment Naming Conventions

Objects generated by the C51 and PL/M-51 compilers are stored in segments which are units of code or data memory. A segment may be relocatable or may be absolute. Each relocatable segment has a type and a name. This section describes the conventions used for naming these segments.

Segment names include a *module_name*. The *module_name* is the name of the source file in which the object is declared and excludes the drive letter, path specification, and file extension. In order to accommodate a wide variety of existing software and hardware tools, all segment names are converted and stored in uppercase.

Each segment name has a prefix (or in case of PL/M-51 a postfix) that corresponds to the memory type used for the segment. The prefix is enclosed in question marks (?). The following is a list of the standard segment name prefixes.

Segment Prefix	Data Type	Description
?PR?	**code**	Executable program code
?CO?	**code**	Constant data in program memory
?XD?	**xdata**	External data memory
?DT?	**data**	Internal data memory
?ID?	**idata**	Indirectly-addressable internal data memory
?BI?	**bit**	Bit data in internal data memory
?BA?	**bdata**	Bit-addressable data in internal data memory
?PD?	**pdata**	Paged data in external data memory

Combining Segments

A segment is a code or data block that is created by the compiler or assembler from your source code. There are two basic types of segments: absolute and relocatable. Absolute segments reside in a fixed memory location. They cannot be moved by the linker. Absolute segments do not have a segment name and will not be combined with other segments. Relocatable segments have a name and a type (as well as other attributes shown in the table below). Relocatable segments with the same name but from different object modules are considered to be parts of the same segment and are called partial segments. The linker/locator combines these partial relocatable segments.

Segments have the following attributes.

Attribute	Description
Name	Each relocatable segment has a name which is used when combining relocatable segments from different program modules. Absolute segments do not have names.
Type	The type identifies the address space to which the segment belongs. The type can be CODE, XDATA, DATA, IDATA, or BIT.
Location Method	The location method specifies the relocation operations that can be performed by the linker/locator. Valid location methods are BITADDRESSABLE, INBLOCK, INPAGE, PAGE, UNIT, and OVERLAYABLE.
Length	The length attribute specifies the length of the segment.
Base Address	The base address specifies the first assigned address of the segment. With absolute segments, the address is assigned by the assembler. With relocatable segments, the address is assigned by the linker/locator.

The above attributes are used to determine how to link, combine, and locate code or data in the segment.

While processing your program modules, the linker/locator produces a table or map of all segments. The table contains name, type, location method, length, and base address of each segment. This table aids in combining partial relocatable segments. All partial segments having the same name are combined by the linker/locator into one single relocatable segment. The linker/locator uses the following rules when combining partial segments.

- All partial segments that share a common name must have the same type (CODE, DATA, IDATA, XDATA or BIT). An error occurs if the types do not correspond.

- The length of the combined segments must not exceed the length of the physical memory area.
- The location method for each of the combined partial segments must correspond.

Absolute segments are not combined with other absolute segments, they are copied directly to the output file.

Locating Segments

After the linker/locator combines partial segments it must determine a physical address for them. The linker/locator processes each physical memory area (internal data, external data, or code space, …) separately. The different memory areas are summarized in the following table.

Memory Area	Length	Address Range	Segment Type
Code	64 KBytes	0000h-FFFFh	CODE
External data	64 KBytes	0000h-FFFFh	XDATA
Internal on-chip data (direct addressable)	128 Bytes	00h-7Fh	DATA
Internal on-chip data (indirect addressable) †	256 Bytes	00h-FFh	IDATA
Bit space in on-chip data †	128 Bits	00-7Fh	BIT

† Refer to the following notes for more information about on-chip RAM.

NOTE
The maximum length of the indirectly addressable data area depends on the 8051 derivative that you are using.

The bit area exists in and overlaps the on-chip data RAM in the byte address range between 20H and 2FH.

The linker/locator places different segments in each of these memory areas. The following sections describe how the linker/locator locates segments in these areas and in which order they are evaluated.

Internal Data Space

Segments that are located in the internal data space include BIT, DATA, IDATA. Memory space for these segments is allocated in the following order:

1. Register Banks
2. Absolute BIT, DATA, and IDATA segments
3. Segments specified with the **PRECEDE** directive on the command line
4. Segments specified with the **BIT** directive on the command line
5. DATA segments that are bit addressable
6. Other relocatable BIT segments
7. Segments specified with the **DATA** directive on the command line
8. Other relocatable DATA segments
9. Segments specified with the **IDATA** directive on the command line
10. Other relocatable IDATA segments with the exception of segments named ?STACK
11. Segments specified with the **STACK** directive on the command line
12. Segments with the name ?STACK and the type IDATA if not specified in any other command line directive

External Data Space

XDATA and PDATA segments are located in the external data space. Memory space for these segments is allocated in the following order:

1. Absolute external data segments
2. Segments specified with the **XDATA** directive on the command line
3. Other relocatable external data segments.

Code Space

Only the CODE segment is located in the code space. Memory space is allocated in the following order:

1. Absolute code segments
2. Segments specified with the **CODE** directive on the command line
3. Other relocatable code segments.

Overlaying Data Memory

The 8051 CPU has a very limited amount of available stack space at run-time. For this reason, local variables and function arguments of C and PL/M-51 routines are stored at fixed memory locations rather than on the stack. By using techniques to overlay the parameters and local variables of C and PL/M-51 functions, the linker/locator attempts to maximize the amount of available space.

To accomplish overlaying, the linker/locator analyzes all references or calls between the various functions. Using this information, the linker/locator can determine precisely which data and bit segments can be overlaid.

You may use the **OVERLAY** and **NOOVERLAY** directives to enable or disable data overlaying. The **OVERLAY** directive is the default and allows for very compact data areas. Use the **NOOVERLAY** directive to disable the segment overlay function.

Resolving External References

External symbols reference addresses in other modules. A declared external symbol must be resolved with a public symbol of the same name. Therefore, for each external symbol, a public symbol must exist in another module.

The linker/locator builds a table of all public and external symbols that it encounters. External references are resolved with public references as long as the names match and the symbol types correspond (for example; DATA, IDATA, XDATA, …).

The linker/locator reports an error if the symbol types of an external and public symbol do not correspond. The linker/locator also reports an error if no public symbol is found for an external reference.

The absolute addresses of the public symbols are resolved after the location of the segments is determined.

Absolute Address Calculation

After the segments are assigned fixed memory locations and external and public references are processed, the BL51 code banking linker/locator calculates the absolute addresses of the relocatable addresses and external addresses. Symbolic debugging information is also updated to reflect the new addresses.

Generating an Absolute Object File

The linker/locator generates the executable target program in Intel OMF-51 absolute object module format. The generated object module may contain debugging information if the linker/locator is so directed. This information facilitates symbolic debugging and testing. You may use the **NODEBUGSYMBOLS**, **NODEBUGPUBLICS**, and **NODEBUGLINES** directives to suppress debugging information in the object file.

The output file generated by the BL51 code banking linker/locator may be loaded by DS51 or an in-circuit emulator, or may be translated by the OC51 Banked Object File Converter and/or the OH51 Object-Hex Converter into an Intel HEX file for use with an EPROM programmer.

Generating a Listing File

The linker/locator generates a listing file that lists information about each step in the link and locate process. This file also contains information about the symbols and segments involved in the linkage. In addition, the following information may be found in the listing file:

- The filenames and other parameters specified on the command line.
- Filenames and module names of all processed modules.
- A memory allocation table which contains the location of the segments, the segment type, the location method, and the segment name. This table may be suppressed by specifying the **NOMAP** directive on the command line.
- The overlay map which shows the structure of the finished program and lists position information for the DATA and BIT function segments. The overlay map also lists all code segments for which OVERLAYABLE BIT and OVERLAYABLE DATA segments exist. You may suppress the overlay map by specifying the **NOMAP** directive on the command line.
- A list of all errors in segments and symbols. The error causes are listed at the end of the listing file.
- A list of all unresolved external symbols. An external symbol is unresolved if no corresponding public symbol exists in another input file. Each

reference to an unresolved external symbol is listed in an error message at the end of the listing file.

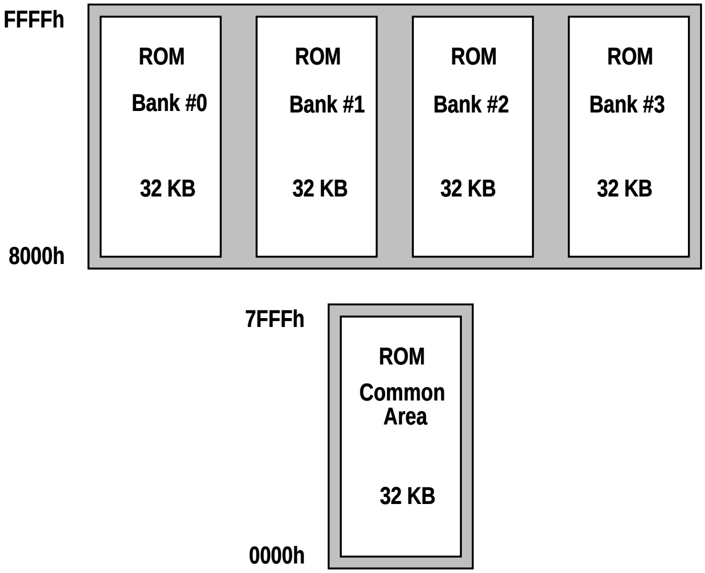
- A symbol table which contains the symbol information from the input files. This information consists of the names of the MODULES, SYMBOLS, PUBLICS, and LINES. LINES are the line numbers produced by a high level language compiler such as the C51 compiler or the PL/M-51 compiler. You may selectively suppress the symbolic information by specifying the **NOSYMBOLS**, **NOPUBLICS**, and **NOLINES** directives on the command line.
- An alphabetically sorted cross reference report of all PUBLIC and EXTERN symbols in which the type of the symbol and the names of the modules are displayed. The first module name is the module in which the PUBLIC symbol is defined. Further module names show the modules in which the EXTERN symbol is defined. If no PUBLIC symbol is present, the message `** UNRESOLVED **` is shown. To produce this cross reference report, specify the **IXREF** directive on the command line.

Errors detected during the execution of the BL51 code banking linker/locator are displayed on the screen as well as at the end of the listing file. A summary of the BL51 code banking linker/locator errors and their causes are described later in this section.

Bank Switching

The 8051 directly supports a maximum of 64 KBytes of code space. The BL51 code banking linker/locator allows 8051 programs to be created that are larger than 64 KBytes by using a technique known as code banking or bank switching. Bank switching involves using extra hardware to select one of a number of code banks all of which will reside at a common physical address.

For example, your hardware design may include one 32K ROM mapped from address 0000h to 7FFFh (known as the common area or common ROM) and four 32K ROMs mapped from code address 8000h to 0FFFFh (known as the code bank ROMs). The code bank ROMs are typically selected using either two port bits or two bits in a memory mapped address in XDATA. One of the four ROMs may then be selected by writing the binary values 00b, 01b, 10b, or 11b to these two bits. The following figure shows the memory structure.



The program code invoked by the BL51 code banking linker/locator to switch or select a particular bank is found in the file **L51_BANK.A51** in the subdirectory **\C51\LIB.** You may alter this file to suit the needs of your particular implementation.

The code banking facility of BL51 is compatible with the C51 compiler and the PL/M-51 compiler program modules. Modules written using either of these two languages can be easily used in code banking applications. No modifications to the original source files are required.

Refer to “Bank Switching Directives” on page 43 for more information on the **BANKx**, **BANKAREA**, and **COMMON** directives and instructions for building code banking programs.

Using RTX51 and RTX51 Tiny

Programs you create that utilize the RTX51 and RTX51 Tiny Real-Time Operating Systems must be linked using the BL51 code banking linker/locator. The **RTX51** and **RTX51TINY** directives enable link-time options that are required to generate RTX51 Full and RTX51 Tiny applications.

Linking Programs with BL51

To invoke the BL51 code banking linker/locator, type **BL51** at the DOS prompt followed by any object modules or directives and press **Enter**. You may include object modules and directives on the command line or you may specify a command response file. Use one of the following command-line formats:

```
BL51 ⟦inputlist⟧ ⟦TO outputfile⟧ ⟦directives⟧
```

or

```
BL51 @commandfile
```

where

inputlist is a list of the object files, separated by commas, for the linker/locator to include in the final absolute object module or banked object module . The files named in the ***inputlist*** can contain both absolute and relocatable program modules which are combined to form the final absolute object module. Additionally, you may force the inclusion of modules from library files by specifying their names in parentheses immediately following the library file name.

outputfile is the name of the absolute object file that the linker/locator creates. If no ***outputfile*** is specified on the command line, the first filename in the input list is used. The basename of the ***outputfile*** is used as base for the **.M51** map file.

directives are commands and parameters that control the operation of the BL51 code banking linker/locator.

commandfile is the name of a command input file that may contain an ***inputlist, outputfile,*** and ***directives***.

The ***inputlist*** uses the following general format:

```
filename ⟦(modulename ⟦, …⟧)⟧ ⟦, …⟧
```

where

filename is the name of an object file created by the C51 compiler or the A51 assembler or a library file created by the LIB51 library manager. The ***filename*** must be specified with its file extension. Object files use the extension **.OBJ**. Library files use the extension **.LIB**.

modulename is the name of an object module in the library file. The ***modulename*** may only be used after the name of a library file. The ***modulenames*** must be specified in parentheses after the filename. Multiple ***modulenames*** may be separated by commas.

Long Command Lines

The invocation line for the BL51 code banking linker/locator may be very long due to the number of specified input files and directives. To enter very long command lines, type the ampersand character (**&**) at the end of a line to indicate that you want to enter more arguments. The BL51 code banking linker/locator prompts you with a double greater than sign (>>) to indicate that you may enter more arguments.

Command Files

In addition to using the ampersand character, you may specify all command-line arguments for the BL51 code banking linker/locator in a command file. This has the same format as a normal command line and may be produced by a text editor. The BL51 code banking linker/locator interprets the first filename preceded by an at sign (**@**) as a command file.

Command-Line Examples

The following examples are proper command lines for the BL51 code banking linker/locator.

```
BL51 C:\MYDIR\PROG.OBJ TO C:\MYDIR\PROG.ABS
```

In this example, only the input file, **C:\MYDIR\PROG.OBJ**, is processed and the absolute object file generated is stored in the output file **C:\MYDIR\PROG.ABS**.

```
BL51 SAMPLE1.OBJ, SAMPLE2.OBJ, SAMPLE3.OBJ &
>> TO SAMPLE.ABS
```

In this example, the files **SAMPLE1.OBJ**, **SAMPLE2.OBJ**, and **SAMPLE3.OBJ** are linked and absolute object file that is generated is stored in the file **SAMPLE.ABS**.

```
BL51 PROG1.OBJ, PROG2.OBJ, UTILITY.LIB
```

In this example, unresolved external symbols are resolved with the public symbols from the library file **UTILITY.LIB**. The modules required from the library are linked automatically. Modules from the library that are not referenced are not included in the generated absolute object file.

```
BL51 PROG1.OBJ, PROG2.OBJ, UTILITY.LIB (FPMUL, FPDIV)
```

In this example, unresolved external symbols are resolved with the public symbols from the library file **UTILITY.LIB**. The modules required from the library are linked automatically. In addition, the **FPMUL** and **FPDIV** modules are included whether or not they are needed. Other modules from the library that are not referenced are not included in the generated absolute object file.

DOS Errorlevel

After linking, the BL51 code banking linker/locator sets the DOS **ERRORLEVEL** to indicate the status of the linking process. Values are listed in the following table.

ERRORLEVEL	Meaning
0	No ERRORS or WARNINGS
1	WARNINGS only
2	ERRORS and possibly also WARNINGS
3	FATAL ERRORS

You can access the **ERRORLEVEL** variable in DOS batch files. Refer to your *DOS User's Guide* for more information about **ERRORLEVEL** or batch files.

Output File

The BL51 code banking linker/locator creates an output file using the input object files that you specify on the command line. The output file is an absolute object file that may be loaded by DS51 for debugging. In addition, you may use the OH51 Object-Hex Converter to create an Intel HEX file from the absolute object file.

Command-Line Directives

Command-line directives may be entered after the output file specification. Multiple directives must be separated by at least one space character (‘ ‘). Each directive may be entered only once on the command line. If a directive is entered twice, the BL51 code banking linker/locator reports an error.

BL51 code banking linker/locator directives fall into one of the following categories:

- Listing File Directives
- Absolute Object File Directives
- Segment Size and Location Directives
- High-Level Language Directives
- Code Banking Directives

The following table lists all BL51 code banking linker/locator directives along with their abbreviations and brief descriptions.

Directive	Abbreviation	Description
BANK*x*	**B*x***	Specifies the starting address and/or segments and/or object modules for code bank *x* (where *x* is a code bank from 0 to 31).
BANKAREA	**BA**	Specifies the address range where the code banks are located.
BIT	**BI**	Locates BIT segments.
CODE	**CO**	Locates CODE segments.
COMMON	**CO**	Specifies the starting address and/or segments and/or object modules to place in the common bank. This directive is essentially the same as the CODE directive.
DATA	**DA**	Locates internal DATA segments.
IDATA	**ID**	Locates internal IDATA segments.
IXREF	**IX**	Directs the BL51 code banking linker/locator to include a cross reference report in the listing file.
NAME	**NA**	Specifies a module name for the absolute object output file.
NOAMAKE		Specifies that AMAKE information is to be excluded from the generated absolute object module.
NODEBUGLINES	**NODL**	Excludes line number information from the absolute object output file.
NODEBUGPUBLICS	**NODP**	Excludes public symbol information from the absolute object output file.

Directive	Abbreviation	Description
NODEBUGSYMBOLS	**NODS**	Excludes local symbol information from the absolute object output file.
NODEFAULTLIBRARY	**NLIB**	Prevents the BL51 code banking linker/locator from including modules from the run-time libraries.
NOLINES	**NOLI**	Prevents the BL51 code banking linker/locator from including line number information in the listing file.
NOMAP	**NOMA**	Prevents the BL51 code banking linker/locator from including a memory map in the listing file.
NOOVERLAY	**NOOL**	Prevents the BL51 code banking linker/locator from overlaying or overlapping local BIT and DATA segments.
NOPUBLICS	**NOPU**	Prevents the BL51 code banking linker/locator from including a list of the public symbols in the listing file.
NOSYMBOLS	**NOSY**	Prevents the BL51 code banking linker/locator from including a list of the local symbols in the listing file.
OVERLAY	**OL**	Directs the BL51 code banking linker/locator to overlay local BIT and DATA segments. Also allows you to specify reference modifications between function segments.
PAGELENGTH	**PL**	Specifies the lines to print on a page in the listing file.
PAGEWIDTH	**PW**	Specifies the number of characters to print on a line in the listing file.
PDATA		Specifies the starting address for PDATA segments.
PRECEDE	**PC**	Locates segments in the register and bit memory areas.
PRINT	**PR**	Specifies the name of the listing file.
RAMSIZE	**RS**	Specifies the size of the on-chip data memory.
REGFILE	**RF**	Specifies the name of the generated file that will contain register usage information.
RTX51		Specifies that the BL51 code banking linker/locator link the application with support for the RTX51 Real-Time Multitasking Operating System.
RTX51TINY		Specifies that BL51 code banking linker/locator link the application with support for the RTX51 Tiny Real-Time Multitasking Operating System.
STACK	**ST**	Locates STACK segments.
XDATA	**XD**	Locates XDATA segments.

The command-line directives are summarized in the following chapter. Refer to “BL51 Directive Reference” on page 57 for an alphabetical listing of the directives complete with descriptions and examples.

Directive Summary

BL51 code banking linker/locator command-line directives fall into one of the following categories.

- Listing File Directives
- Output File Directives
- Segment Size and Location Directives
- High-Level Language Directives
- Code Bank Switching Directives
- RTX51 Directives

The following sections describe these categories and the directives they encompass.

Listing File Directives

The BL51 code banking linker/locator generates a listing file that contains information about the link/locate process. This file is sometimes referred to as a map file. The following directives control the filename, format, and information that is included in the listing file.

IXREF
NOLINES
NOMAP
NOPUBLICS
NOSYMBOLS
PAGELENGTH
PAGEWIDTH
PRINT
PUBLICS
SYMBOLS

Each of these directives is described below.

PRINT

By default, the listing file is given the basename of the output file specified on the command line along with the extension **.M51**. However, you may use the **PRINT** directive to specify the name of the listing file. For example, the following command line:

```
BL51 MYPROG.OBJ TO MYPROG.ABS PRINT(OUTPUT.MAP)
```

directs the BL51 code banking linker/locator to write the listing information to the file **OUTPUT.MAP**. You may specify `PRINT(LPT1:)` to direct the BL51 code banking linker/locator to send the list file to the printer.

PAGELENGTH & PAGEWIDTH

Use the **PAGELENGTH** and **PAGEWIDTH** directives to control the number of lines per page and the number of characters per line respectively. You must specify these numbers in parentheses following the directive. The following example instructs the BL51 code banking linker/locator to generate the listing file with 50 lines per page and 100 characters per line.

```
BL51 PROG.OBJ TO PROG.ABS PAGELENGTH(50) PAGEWIDTH(100)
```

IXREF

The **IXREF** directive instructs the BL51 code banking linker/locator to include a cross reference report in the listing file. A cross reference report lists symbols, the area of memory in which they are located (for example, CODE, XDATA, DATA, IDATA, or BIT), and the source modules in which they are accessed.

You may optionally exclude compiler-generated symbols by specifying the **NOGENERATED** argument in parentheses immediately following the **IXREF** directive. You may use **NOGN** as an abbreviation for **NOGENERATED**.

You may optionally exclude symbol contained within libraries by specifying the **NOLIBRARIES** argument in parentheses following the **IXREF** directive. You may use **NOLI** as an abbreviation for **NOLIBRARIES**.

The following examples show you how to use the **IXREF** directive.

```
BL51 SAMPLE1.OBJ, SAMPLE2.OBJ, SAMPLE3.OBJ IXREF

BL51 SAMPLE1.OBJ, SAMPLE2.OBJ, SAMPLE3.OBJ IXREF(NOGENERATED)

BL51 SAMPLE1.OBJ, SAMPLE2.OBJ, SAMPLE3.OBJ IXREF(NOLIBRARIES)
```

NOMAP

The **NOMAP** directive prevents the BL51 code banking linker/locator from including the memory map in the listing file.

Example:

```
BL51 MYPROG.OBJ NOMAP
```

NOSYMBOLS

The **NOSYMBOLS** directive prevents the BL51 code banking linker/locator from including this table in the listing file.

Example:

```
BL51 MYPROG.OBJ NOSYMBOLS
```

NOPUBLICS

The **NOPUBLICS** directives prevents the BL51 code banking linker/locator from including this table in the listing file.

Example:

```
BL51 MYPROG.OBJ NOPUBLICS
```

NOLINES

The **NOLINES** directives prevents the BL51 code banking linker/locator from including line number information in the listing file. Line number information is generated for debugging purposes. The BL51 code banking linker/locator can generate a table of line numbers and addresses for source modules in your program.

```
BL51 MYPROG.OBJ NOLINES
```

Example Listing File

The following example includes all optional sections of the listing file.

```
BL51 BANKED LINKER / LOCATER  BL51 V3.x                     DATE  01/19/93   PAGE
1



MS-DOS BL51 LINKER / LOCATER  BL51 V3.x, INVOKED BY:
BL51 MEASURE.OBJ, MCOMMAND.OBJ, GETLINE.OBJ XDATA (4000H) IX

MEMORY MODEL: SMALL WITH FLOATING POINT ARITHMETIC
```

The listing file shows the command line that invoked the linker.

```
INPUT MODULES INCLUDED:
  MEASURE.OBJ (MEASURE)
  MCOMMAND.OBJ (MCOMMAND)
  GETLINE.OBJ (GETLINE)
  C:\C51\LIB\C51FPS.LIB (?C_FPADD)
  C:\C51\LIB\C51FPS.LIB (?C_FPMUL)
  C:\C51\LIB\C51FPS.LIB (?C_FPDIV)
  C:\C51\LIB\C51FPS.LIB (?C_FPCMP)
  C:\C51\LIB\C51FPS.LIB (?C_FCAST)
.
.
.
  C:\C51\LIB\C51S.LIB (?C_LSTXDATA)
  C:\C51\LIB\C51S.LIB (?C_LSTPDATA)
  C:\C51\LIB\C51S.LIB (?C_ISTACKD)
```

Object modules that were included are listed at the beginning of the listing.

```
LINK MAP OF MODULE:  MEASURE (MEASURE)


TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
-----------------------------------------------------

* * * * * *     D A T A   M E M O R Y   * * * * * * *
REG     0000H     0008H     ABSOLUTE     "REG BANK 0".
REG     0008H     0008H     ABSOLUTE     "REG BANK 1"
DATA    0010H     0010H     UNIT         ?C_LIB_DATA
DATA    0020H     0001H     BIT_ADDR     ?C_LIB_DBIT
BIT     0021H.0   0000H.3   UNIT         ?BI?MEASURE
BIT     0021H.3   0000H.1   UNIT         ?BI?GETCHAR

* * * * * * *  X D A T A   M E M O R Y  * * * * * * *
        0000H     4000H                  *** GAP ***
XDATA   4000H     1FF8H     UNIT         ?XD?MEASURE

* * * * * * *   C O D E   M E M O R Y   * * * * * * *
CODE    0000H     0003H     ABSOLUTE
CODE    0003H     0005H     UNIT         ?PR?GETCHAR?UNGETCHAR
        0008H     0003H                  *** GAP ***
CODE    000BH     0003H     ABSOLUTE
CODE    000EH     005BH     UNIT         ?PR?SAVE_CURRENT_MEASUREMENTS?MEASUR
CODE    0069H     00D3H     UNIT         ?PR?TIMER0?MEASURE
CODE    013CH     008BH     UNIT         ?PR?_READ_INDEX?MEASURE
CODE    01C7H     0035H     UNIT         ?PR?CLEAR_RECORDS?MEASURE
CODE    1BD2H     0011H     UNIT         ?PR?GETCHAR?GETCHAR
CODE    1BE3H     0016H     UNIT         ?PR?_ISSPACE?ISSPACE
CODE    1BF9H     0018H     UNIT         ?PR?_TOUPPER?TOUPPER
.
.
.
```

The memory map is included after the object modules. You can disable the memory map using the NOMAP directive.

```
OVERLAY MAP OF MODULE:   MEASURE (MEASURE)


SEGMENT                                     BIT-GROUP      DATA-GROUP
  +--> CALLED SEGMENT                    START   LENGTH   START   LENGTH
------------------------------------------------------------------------
?PR?TIMER0?MEASURE                       -----   -----    -----    -----
  +--> ?PR?SAVE_CURRENT_MEASUREMENTS?MEASURE
  +--> ?C_LIB_CODE

?PR?SAVE_CURRENT_MEASUREMENTS?MEASURE    -----   -----    -----    -----
  +--> ?C_LIB_CODE

?C_C51STARTUP                            -----   -----    -----    -----
  +--> ?PR?MAIN?MEASURE
  +--> ?C_INITSEG

?PR?MAIN?MEASURE                         -----   -----    003CH    0003H
  +--> ?PR?CLEAR_RECORDS?MEASURE
```

An overlay map is listed after the memory map. The overlay map shows the call tree of your application.

```
  +--> ?CO?MEASURE
  +--> ?PR?PRINTF?PRINTF
  +--> ?PR?_GETLINE?GETLINE
  +--> ?PR?_TOUPPER?TOUPPER
  +--> ?PR?_READ_INDEX?MEASURE
  +--> ?PR?_GETKEY?_GETKEY
  +--> ?C_LIB_CODE
  +--> ?PR?MEASURE_DISPLAY?MCOMMAND
  +--> ?PR?_SET_TIME?MCOMMAND
  +--> ?PR?_SET_INTERVAL?MCOMMAND

?PR?CLEAR_RECORDS?MEASURE                 -----   -----    -----    -----
  +--> ?C_LIB_CODE

?PR?PRINTF?PRINTF                        0021H.4  0001H.1   004BH    001CH
  +--> ?C_LIB_CODE
  +--> ?PR?PUTCHAR?PUTCHAR

?PR?_GETLINE?GETLINE                      -----   -----    003FH    0004H
  +--> ?PR?_GETKEY?_GETKEY
  +--> ?PR?PUTCHAR?PUTCHAR
.
.
.
SYMBOL TABLE OF MODULE:  MEASURE (MEASURE)

VALUE           TYPE          NAME
----------------------------------

-------         MODULE        MEASURE
C:0000H         SYMBOL        _ICE_DUMMY_
```

The symbol table lists public, local, and line number information.

```
B:00C8H.0       PUBLIC        T2I0
B:00C8H.1       PUBLIC        T2I1
B:00B0H.4       PUBLIC        T0
B:00D0H.6       PUBLIC        AC
D:00E8H         PUBLIC        P4
B:00B0H.5       PUBLIC        T1
D:00F8H         PUBLIC        P5
B:00D8H.7       PUBLIC        BD
D:0023H         PUBLIC        current
.
.
.
```

You can use the NOPUBLICS directiveto exclude public symbols from the listing.

```
D:000FH         SYMBOL        i
C:0076H         LINE#         104
C:0079H         LINE#         105
C:007CH         LINE#         106
.
.
.
```

You can use the NOSYMBOLS directive to exclude local symbols from the listing.

```
C:00D0H         LINE#         125
C:00D0H         LINE#         126
C:00D0H         LINE#         128
C:00E5H         LINE#         129
C:00E9H         LINE#         131
C:00F1H         LINE#         132
C:00F3H         LINE#         134
.
.
.
```

You can use the NOLINES directive to exclude line number information from the listing.

```
INTER-MODULE CROSS-REFERENCE LISTING
------------------------------------


NAME . . . . . . . . . . . USAGE   MODULE NAMES
-----------------------------------------------

?C_ATOFFIRSTCALL . . . . . BIT;    ?C_ATOF  SCANF
```

The IXREF directive instructs L51 toinclude a cross reference table.

```
?C_CASTF . . . . . . . . . CODE;   ?C_CASTF  MCOMMAND
?C_CASTF2. . . . . . . . . CODE;   ?C_CASTF
?C_CCASE . . . . . . . . . CODE;   ?C_CCASE  PRINTF  SCANF
?C_CHARLOADED. . . . . . . BIT;    GETCHAR  UNGETC
?C_CLDOPTR . . . . . . . . CODE;   ?C_CLDOPTR  PRINTF  SCANF
?C_CLDPTR. . . . . . . . . CODE;   ?C_CLDPTR  PRINTF  SCANF
?C_COPY. . . . . . . . . . CODE;   ?C_COPY  MCOMMAND  MEASURE
.
.
.
```

Output File Directives

The linker/locator generates either absolute object files or banked object files. Banked object files must be converted, by the OC51 Banked Object File Converter, into absolute object files (one for each bank).

Absolute object files contain no relocatable or external references and can be converted by the OH51 Object-Hex Converter into Intel HEX files. Intel HEX files may be directly loaded into an emulator or EPROM programmer. The following directives control the module name, as well as debugging and source module information that may be included in the absolute object file.

NAME
NOAMAKE
NODEBUGLINES
NODEBUGPUBLICS
NODEBUGSYMBOLS

These directives are described in the following sections.

NAME

You can specify a module name for the absolute object module that the linker/locator generates using the **NAME** directive. The **NAME** directive may be accompanied by the module name (enclosed in parentheses) that you want to assign it. In the following,

```
BL51 MYPROG.OBJ TO MYPROG.ABS NAME(BIGPROG)
```

BIGPROG is the module name stored in the object file. If no module name is specified with the **NAME** directive, the name of the first input module is used for the module name.

NOTE
*The module name specified with the **NAME** directive is not the filename of the*

absolute object file. The module name is stored in the object module file and may be accessed only by a program that reads the contents of that file.

NOAMAKE

By default, the BL51 code banking linker/locator generates object modules that include source file information records. These records contain time and date information for the source file and its include files.

Use the **NOAMAKE** directive to prevent the BL51 code banking linker/locator from including these record types in the generated object module. This may be useful if you have conversion programs that cannot recognize these record formats.

NODEBUGLINES

The BL51 code banking linker/locator includes line number information in the absolute object file that it generates. Line number information are the line numbers of your source modules along with the code addresses for each line. When you debug your program using an in-circuit emulator or a simulator, you can step through your program line by line. This is often referred to as source level debugging.

The **NODEBUGLINES** directive directs the BL51 code banking linker/locator to exclude line number information from the object file. This directive is used as follows:

```
BL51 MYPROG.OBJ NODEBUGLINES
```

You may wish to exclude line number information when you are creating your final production object file.

NOTE
In order for the BL51 code banking linker/locator to include debugging information in the output object file, that information must already be available in the input object files. Refer to the A51 User’s Guide and C51 User’s Guide for information on including debugging information in the object files.

NODEBUGPUBLICS

The BL51 code banking linker/locator can includes public symbols in the generated absolute object file. The public symbols information can be used by simulators and in-circuit emulators to display values and address information for public variables when debugging your program.

The **NODEBUGPUBLICS** directive directs the BL51 code banking linker/locator to exclude public symbol information from the object file. This directive is used as follows:

```
BL51 MYPROG.OBJ NODEBUGPUBLICS
```

You may wish to exclude public symbol debugging information when you are creating your final production object file.

NOTE
In order for the BL51 code banking linker/locator to include debugging information in the output object file, that information must already be available in the input object files. Refer to the A251/ A51 User's Guide and C51 User's Guide for information on including debugging information in the object files.

NODEBUGSYMBOLS

The BL51 code banking linker/locator includes local symbol debugging information in the absolute object file. Typically, you may use this information with a simulator or in-circuit emulator to display the values of local symbols used in your program.

The **NODEBUGSYMBOLS** directive directs the BL51 code banking linker/locator to exclude local symbol information from the object file. This directive is used as follows:

```
BL51 MYPROG.OBJ NODEBUGSYMBOLS
```

You may wish to exclude local symbol debugging information when you are creating your final production object file.

NOTE
In order for the BL51 code banking linker/locator to include debugging information in the output object file, that information must already be available

in the input object files. Refer to the A251 / A51 User's Guide and C51 User's Guide for information on including debugging information in the object files.

Segment Size and Location Directives

The BL51 code banking linker/locator allows you to specify the size of the different memory areas or segments, the order of the segments within the different memory areas, and the location or absolute memory address of different segments. These segment manipulations are performed using the following directives.

BIT
CODE
DATA
IDATA
PDATA
PRECEDE
RAMSIZE
STACK
XDATA

The BL51 code banking linker/locator locates segments in three memory areas—Internal Data, External Data, or Code—and follows a predefined order of precedence. Note that the standard allocation algorithms usually produce the best workable solution without requiring you to enter any additional information on the command line. However, the directives described in this chapter allow you to more closely control the location of segments within the different memory spaces.

RAMSIZE

The BL51 code banking linker/locator links and locates your program assuming that there are 128 bytes of internal data memory available in your target processor. This is true of most of the 8051 derivatives; however, a number of derivatives have more or less than 128 bytes of memory.

Use the **RAMSIZE** directive to specify the number of bytes of internal data memory in your target 8051 derivative. The number of bytes of internal data memory must be specified enclosed within parentheses. For example:

```
BL51 MYPROG.OBJ RAMSIZE(256)
```

This example links `MYPROG.OBJ` and specifies that there are 256 bytes of internal memory that may be allocated by the linker.

The size of the internal data memory may be a number between 64 and 256. Values outside this range generate a linker error.

BIT

The **BIT** directive lets you specify:

- The starting address for segments placed in the bit-addressable internal data space
- The order of segments within the bit-addressable internal data space
- The absolute memory location of segments in the bit-addressable internal data space.

Addresses that you specify with the **BIT** directive are bit addresses. They are not byte addresses. In the 8051, bit addresses 00h through 7Fh reference bits in internal data memory bytes from byte address 20h to 2Fh (16 bytes of 8 bits each, 16 × 8 = 128 = 80h). Bit addresses that are evenly divisible by 8 reference the low-order bit for its corresponding byte and are also considered to be aligned on a byte border. A DATA segment that is bit-addressable can be located with the **BIT** directive; however, the specified bit address must lie on a byte boundary. The bit address must be evenly divisible by 8.

To specify the starting address for segments stored in bit-addressable internal data memory, you must include the starting address in parentheses with the **BIT** directive on the command line, as shown in the following examples.

```
BL51 MYPROG.OBJ BIT(48)
```

or

```
BL51 MYPROG.OBJ BIT(30h)
```

The first example specifies that relocatable BIT segments be located at or after bit address 48 decimal (30 hex) which is equivalent to byte address 26 hex in the internal data memory. The second example specifies that relocatable BIT segments be located at or after bit address 30 hex.

To specify the order for segments stored in bit-addressable internal data memory, you must include the names of the segments, separated by commas, in parentheses with the **BIT** directive on the command line, as shown in the following example.

```
BL51 MYPROG.OBJ,A.OBJ,B.OBJ,C.OBJ BIT(?DT?A,?DT?B,?DT?C)
```

This example places the **?DT?A**, **?DT?B**, and **?DT?C** segments at the beginning of the bit-addressable internal data memory.

You may also specify the bit address for the segments you specify with the **BIT** directive, for example:

```
BL51 MYPROG.OBJ,A.OBJ,B.OBJ BIT(?DT?A(28h),?DT?B(30h))
```

This example places the **?DT?A** and **?DT?B** segments at **28h** and **30h**, respectively, in the bit-addressable internal data memory.

DATA

The **DATA** directive allows you to specify the starting address for segments placed in the directly–addressable internal data space, the order of segments within the directly–addressable internal data space, and the absolute memory location of segments in the directly–addressable internal data space.

To specify the starting address for segments stored in directly–addressable internal data memory, you must include the starting address enclosed within parenthesis with the **DATA** directive on the command line. For example:

```
BL51 MYPROG.OBJ DATA(48)
```

or

```
BL51 MYPROG.OBJ DATA(30h)
```

The first example above specifies that relocatable DATA segments be located at or after address 48 decimal (30 hex) in the internal data memory. The second example specifies that relocatable DATA segments be located at or after address 30 hex.

To specify the order for segments stored in directly–addressable internal data memory, you must include the names of the segments separated by commas and enclosed within parenthesis with the **DATA** directive on the command line. For example:

```
BL51 MYPROG.OBJ,A.OBJ,B.OBJ,C.OBJ DATA(?DT?A,?DT?B,?DT?C)
```

This example will place the **?DT?A**, **?DT?B**, and **?DT?C** segments at the beginning of the directly–addressable internal data memory.

You can also specify the memory location of the segments you specify with the **DATA** directive. For example:

```
BL51 MYPROG.OBJ,A.OBJ,B.OBJ DATA(?DT?A(28h),?DT?B(30h))
```

This example will place the **?DT?A** and **?DT?B** segments at **28h** and **30h** in the directly–addressable internal data memory respectively.

IDATA

The **IDATA** directive lets you specify:

- The starting address for segments placed in the indirectly-addressable internal data space
- The order of segments within the indirectly-addressable internal data space
- The absolute memory location of segments in the indirectly-addressable internal data space.

To specify the starting address for segments stored in indirectly-addressable internal data memory, you must include the starting address in parentheses with the **IDATA** directive on the command line, for example:

```
BL51 MYPROG.OBJ IDATA(64)
```

or

```
BL51 MYPROG.OBJ IDATA(40h)
```

The first example specifies that relocatable IDATA segments be located at or after address 64 decimal (40 hex) in the internal data memory. The second example specifies that relocatable IDATA segments be located at or after address 40 hex.

To specify the order for segments stored in indirectly-addressable internal data memory, you must include the names of the segments, separated by commas, in parentheses with the **IDATA** directive on the command line, for example:

```
BL51 MYPROG.OBJ,A.OBJ,B.OBJ,C.OBJ IDATA(?ID?A,?ID?B,?ID?C)
```

This example places the **?ID?A**, **?ID?B**, and **?ID?C** segments at the beginning of the indirectly-addressable internal data memory.

You may also specify the location of the segments you specify with the **IDATA** directive, for example:

```
BL51 MYPROG.OBJ,A.OBJ,B.OBJ IDATA(?ID?A(30h),?ID?B(40h))
```

This example places the **?ID?A** and **?ID?B** segments at **30h** and **40h**, respectively, in the indirectly-addressable internal data memory.

PRECEDE

The **PRECEDE** directive allows you to specify segments that lie in the internal data memory that should precede all other segments in that memory space. Segments that you specify with this directive will be located after the BL51 code banking linker/locator has located register banks and any absolute BIT, DATA, and IDATA segments that may exist in your program.

You specify segment names with the **PRECEDE** directive on the command line. Segment names must be separated by commas and must be enclosed in parentheses immediately following the **PRECEDE** directive, for example:

```
BL51 MYPROG.OBJ,A.OBJ,B.OBJ PRECEDE(?DT?A,?DT?B)
```

The segments that you specify are located at the lowest available memory location in the internal data memory in the order that you specify. You may also specify the memory location of the segments you specify with the **PRECEDE** directive, for example:

```
BL51 MYPROG.OBJ,A.OBJ,B.OBJ PRECEDE(?DT?A(09h),?DT?B(13h))
```

This example places the **?DT?A** and **?DT?B** segments at **09h** and **13h**, respectively, in the internal data memory if it is possible to do so

STACK

Use the **STACK** directive to specify which segments are to be located in the uppermost IDATA memory space in internal data memory. The segments you specify with this directive will follow all other segments in the internal data memory space.

You specify segment names with the **STACK** directive on the command line. Segment names must be separated by commas and must be enclosed in parentheses immediately following the **STACK** directive, for example:

```
BL51 MYPROG.OBJ,A.OBJ,B.OBJ STACK(?DT?A,?DT?B)
```

The segments that you specify are located at the highest available memory location in the internal data memory in the order that you specify. You can also specify the memory location of the segments you specify, for example:

```
BL51 MYPROG.OBJ,A.OBJ,B.OBJ STACK(?DT?A(69h),?DT?B(73h))
```

This example places the `?DT?A` and `?DT?B` segments at `69h` and `73h`, respectively, in the internal data memory if it is possible to do so

The C51 compiler and the PL/M-51 compiler both generate a stack segment called ?STACK which is automatically located at the top of the internal data memory. The 8051 stack pointer is initialized by the startup code to point to this location. All return addresses and data that are pushed are stored in this memory area. It is not necessary to specifically locate stack segments if you are using only C or PL/M-51. The **STACK** directive is usually used with assembly programs in which there might be a number of stack segments.

NOTE
*You should use extreme caution when relocating the ?STACK segment using the **STACK** directive. This operation can easily result in a target program that will not run and that will corrupt system variables.*

CODE

The **CODE** directive allows you to specify:

- The starting address for segments placed in the code memory space
- The order of segments within the code memory space
- The absolute memory location of segments in the code memory space.

To specify the starting address for segments stored in the code space, you must include the starting address in parentheses with the **CODE** directive on the command line, for example:

```
BL51 MYPROG.OBJ CODE(200)
```

or

```
BL51 MYPROG.OBJ CODE(4000h)
```

The first example specifies that relocatable segments in code memory be located at or after address 200 decimal (C8 hex) in the code space. The second example specifies that relocatable segments in code memory be located at or after address 4000 hex.

To specify the order for segments in the code space, you must include the names of the segments, separated by commas, in parentheses with the **CODE** directive on the command line, for example:

```
BL51 MYPROG.OBJ CODE(?PR?FUNC1?MYPROG,?PR?FUNC2?MYPROG)
```

This example places the **`?PR?FUNC1?MYPROG`** and **`?PR?FUNC2?MYPROG`** segments at the beginning of the code memory. These segments contain the C functions **`func1`** and **`func2`**, respectively.

You may also specify the memory location of the segments you specify with the **CODE** directive, for example:

```
BL51 MYPROG.OBJ &

CODE(?PR?FUNC1?MYPROG(1000h),?PR?FUNC2?MYPROG(2000h))
```

This example places the **`?PR?FUNC1?MYPROG`** and **`?PR?FUNC2?MYPROG`** segments at **`1000h`** and **`2000h`**, respectively, in the code space.

XDATA

The **XDATA** directive allows you to specify:

- The starting address for segments placed in the external data space
- The order of segments within the external data space
- The absolute memory location of segments in the external data space.

To specify the starting address for data stored in the external memory space, you must include the starting address in parentheses with the **XDATA** directive on the command line, for example:

```
BL51 MYPROG.OBJ XDATA(100)
```

or

```
BL51 MYPROG.OBJ XDATA(1000h)
```

The first example specifies that relocatable segments in the external data memory be located at or after address 100 decimal (64 hex) in the external data memory. The second example specifies that relocatable segments in external data memory be located at or after address 1000 hex.

To specify the order for segments in the external data memory, you must include the names of the segments, separated by commas, in parentheses with the **XDATA** directive on the command line, for example:

```
BL51 MYPROG.OBJ,A.OBJ,B.OBJ,C.OBJ XDATA(?XD?A,?XD?B,?XD?C)
```

This example places the **?XD?A**, **?XD?B**, and **?XD?C** segments at the beginning of the external data memory.

You may also specify the location of the segments you specify with the **XDATA** directive, for example:

```
BL51 MYPROG.OBJ,A.OBJ,B.OBJ XDATA(?XD?A(100h),?XD?B(200h))
```

This example places the **?XD?A** and **?XD?B** segments at **100h** and **200h**, respectively, in the external data memory.

PDATA

The **PDATA** directive allows you to specify the starting address, in external data memory, for PDATA segments. You must enter the starting address immediately following the **PDATA** directive on the command line. The address must be enclosed in parentheses, for example:

```
BL51 MYPROG.OBJ PDATA(8000h)
```

This example specifies that PDATA segments are to be located starting at address 8000 hex in the external data memory.

In addition to specifying the starting address for PDATA segments on the linker command line, you must also modify the startup code stored in **STARTUP.A51** to indicate that PDATA segments are located at 8000h. Refer to the *C51 User's Guide* for more information about PDATA and COMPACT model programming.

High-Level Language Directives

The BL51 code banking linker/locator provides control over aspects of the output file that have to do with high-level languages like C and PL/M-51. You can control whether or not the BL51 code banking linker/locator includes modules from the run-time library and whether or not the BL51 code banking linker/locator overlays the local variable areas of C and PL/M-51 functions. The directives **NODEFAULTLIBRARY**, **NOOVERLAY**, **OVERLAY**, and **REGFILE** are available for these applications.

NODEFAULTLIBRARY

By default, the BL51 code banking linker/locator includes modules from the run-time libraries that are referenced by your C and PL/M-51 programs.

The run-time libraries may be stored in any subdirectory as long as they are referenced by the **C51LIB** DOS environment variable. This variable can be set by typing the following DOS command at the command prompt:

```
SET C51LIB=C:\C51\LIB
```

This command defines the subdirectory in which the library files are located. This makes it unnecessary for library files to be located in the same subdirectory as the object files for your program. If the **C51LIB** environment variable is not defined, the BL51 code banking linker/locator searches for the library files in the current directory only.

The library file is chosen based on the memory model and floating-point requirements of the object files. The following libraries are automatically added their uses.

Library File	Description
C51S.LIB	Small model library without floating-point arithmetic
C51FPS.LIB	Small model floating-point arithmetic library
C51C.LIB	Compact model library without floating-point arithmetic
C51FPC.LIB	Compact model floating-point arithmetic library
C51L.LIB	Large model library without floating-point arithmetic
C51FPL.LIB	Large model floating-point arithmetic library
PLM51.LIB	Library for Intel PL/M-51.

You may use the **NODEFAULTLIBRARY** directive to prevent the BL51 code banking linker/locator from including modules from these run-time libraries, for example:

```
BL51 MYPROG.OBJ NODEFAULTLIBRARY
```

NOOVERLAY

Because of the limited amount of stack space available on the 8051, local variables and function arguments of C and PL/M-51 routines are stored at fixed memory locations rather than on the stack. Normally, the BL51 code banking linker/locator attempts to overlay this memory by analyzing your program and creating a call tree of the routines that it finds.

This technique usually works very well and provides a more efficient use of memory than a conventional stack frame would. However, in certain situations, this can be undesirable.

You may use the **NOOVERLAY** directive to disable overlay analysis and implementation. When this directive is specified on the command line, the BL51 code banking linker/locator does not overlay variables and function argument data space. The **NOOVERLAY** directive is specified as follows:

```
BL51 MYPROG.OBJ NOOVERLAY
```

OVERLAY

The 8051 CPU has a very limited amount of available stack space at run-time. For this reason, local variables and function arguments of C and PL/M-51 routines are stored at fixed memory locations rather than on the stack.

The BL51 code banking linker/locator attempts to overlay this memory by analyzing your program and creating a call tree of the function references between the various code segments. The appropriate data and bit segments are determined by standard segment naming conventions. It is assumed that the segment names and the implied memory type extensions are the same. Therefore, segments used in your programs should be constructed according to the following rules.

Segment Type	C51 Segment Name	PL/M-51 Segment Name
CODE	?PR?*functionname*?*modulename*	?*modulename*?PR
BIT	?BI?*functionname*?*modulename*	?*modulename*?BI
DATA	?DT?*functionname*?*modulename*	?*modulename*?DT
IDATA	?ID?*functionname*?*modulename*	—
XDATA	?XD?*functionname*?*modulename*	—
PDATA	?PD?*functionname*?*modulename*	—

NOTE
Unless you are writing and interfacing assembly routines to C or to PL/M-51, you do not need to be concerned with these segment naming conventions.

The memory type of the segment names is determined by the prefixes and extensions ?PR, ?BI, ?DT, ?XD, ?ID, and ?PD. Each BIT and DATA segment should contain the OVERLAYABLE attribute.

The C51 and PL/M-51 compilers automatically define BIT and DATA segments according to these rules. However, if you use overlayable segments in your assembly modules, you must follow these naming conventions. Refer to the *A251 / A51 User's Guide* for information on how to declare segments.

Data and bit segments are overlaid under the following conditions:

- No references or calls may exist between the related code segments. During the analysis procedure of the BL51 code banking linker/locator, the direct level, as well as references through other code segments, are considered.
- The code segments may be invoked by only one of the following program types: main or interrupt.
- The segment definitions must have been specified according to the previous rules.

Typically, the BL51 code banking linker/locator analyzes your programs and generates overlay information that is accurate. However, in some instances the analysis performed by the BL51 code banking linker/locator is ineffective. This occurs with indirectly called functions through function pointers and functions that are called by both the main program and an interrupt function.

In these cases, you may use the **OVERLAY** directive to control the references that the BL51 code banking linker/locator uses in its overlay analysis. The **OVERLAY** directive may be specified a number of times in the command for each reference.

The general format of the overlay parameter is as follows:

```
OVERLAY (sfname {! | ~} sfname ⟦, …⟧)
```

or

```
OVERLAY (sfname {! | ~} (sfname, sfname ⟦, …⟧) ⟦, …⟧)
```

or

```
OVERLAY (sfname ! *)
```

or

```
OVERLAY (* ! sfname)
```

where

sfname is a segment name or function name of a C function.

! adds an additional call in the reference listing.

~ deletes a call from the reference listing.

***** is used to add roots or disable segment overlaying.

Each of the forms of the **OVERLAY** directive are described below.

OVERLAY

The **OVERLAY** directive, when specified without any arguments, instructs the BL51 code banking linker/locator to automatically determine code references between modules. This requires that no indirect calls are present in the program modules. The external and public information stored in each input file is used to generate this information.

ʼ OVERLAY (* ! sfname)

The OVERLAY directive can be used to specify a new root for a segment or function name. BL51 handles these functions including their call trees as independent programs. Adding roots to an application is useful when real-time operating systems are used. For example:

```
BL51 SAMPLE.OBJ OVERLAY (* ! TASK0, TASK1)
```

In this example the functions **TASK0** and **TASK1** are handled as independent program roots.

- **OVERLAY (sfname ! *)**

The **OVERLAY** directive may be specified with a segment or function name that is to be excluded from the overlay analysis and processed in a normal fashion. This has no influence on the overlay evaluation of other segments, for example:

```
BL51 SAMPLE.OBJ OVERLAY (FUNC1 ! *)
```

In this example, **FUNC1** is excluded from local segment overlaying.

- **OVERLAY (sfname ! sfname1)**
  OVERLAY (sfname ! (sfname1, sfname2))

The **OVERLAY** directive may be used to add references to the specified segments or functions. The first segment name specified is added to subsequent segments, for example:

```
BL51 CMODUL1.OBJ OVERLAY (FUNC1 ! (FUNC2, FUNC3))
```

In this example, references to the function **FUNC1** are added to **FUNC2** and **FUNC3** for the overlay analysis procedure.

- **OVERLAY (sfname ~ sfname1)**
  OVERLAY (sfname ~ (sfname1, sfname2))

The **OVERLAY** directive may be used to delete or remove references between segments or functions. References to subsequent segments specified in the command line are removed from the first segment name specified, for example:

```
BL51 MAINMOD.OBJ, TEXTOUT.OBJ &
    OVERLAY (FUNC1 ~ ?CO?MAINMOD, FUNC2 ~ ?CO?MAINMOD)
```

In this example, references to the **?CO?MAINMOD** segment are deleted from **FUNC1** and **FUNC2**.

OVERLAY Examples

In most cases, the overlay algorithm works correctly without any adjustments. However, in some instances when the overlay algorithm cannot determine the structure of your program, you must adjust function references with the

OVERLAY directive. This is the case when you use function pointers in your program.

Using the **OVERLAY** directive is easy when you know the structure of your program. The program structure is reflected in the segments listed in the overlay map of the listing file. If you are in doubt about whether certain segments should be overlaid or not, you may disable overlaying of those segments. Segment overlaying can be disabled with the following C51 compiler and BL51 code banking linker/locator options:

- You can invoke the C51 compiler with the **OPTIMIZE (1)** option to disable data overlaying for a whole module.
- You can invoke the BL51 code banking linker/locator with the **OVERLAY (*sfname* ! *)** option to disable data overlaying for ***funcname*** function.
- You can invoke the BL51 code banking linker/locator with the **NOOVERLAY** option to disable data overlaying for the entire application.

The following application examples show situations where the **OVERLAY** directive is required to correct the program structure. In general, a modification of the references (calls) is required in the following cases:

- When a pointer to a function is passed or returned as function argument.
- When a pointer to a function is contained in initialized variables.

Example 1: Using a Pointer to a Function as Function Argument

In the following example **`indirectfunc1`** and **`indirectfunc2`** are indirectly called through a function pointer in **`execute`**. The value of the function pointer is passed in **`main`**. Thus the linker/locator detects that **`main`** calls **`indirectfunc1`** and **`indirectfunc2`**, though the actual function call is executed by **`execute`**.

Following is a program listing for this example.

```
.
.
.
bit indirectfunc1 (void)  {   /* indirect function 1 */
  unsigned char n1, n2;
  return (n1 < n2);
}

bit indirectfunc2 (void)  {   /* indirect function 2 */
  unsigned char a1, a2;
  return ((a1 - 0x41) < (a2 - 0x41));
}


void execute (bit (*fct) ())  {      /* sort routine */
  unsigned char i;
  for (i = 0; i < 10; i++)  {
    if (fct ())  i = 10;
  }
}

void main (void)  {

  if (SWITCH)                  /* switch: defines function */
    execute (indirectfunc1);
  else
    execute (indirectfunc2);
}
.
.
.
```

The following listing file shows the overlay map for the program before making adjustments with the **OVERLAY** directive.

```
OVERLAY MAP OF MODULE:   OVL1 (OVL1)

SEGMENT                          BIT-GROUP           DATA-GROUP
  +--> CALLING SEGMENT         START  LENGTH         START  LENGTH
------------------------------------------------------------------
?C_C51STARTUP                  -----   -----         -----   -----
  +--> ?PR?MAIN?OVL1

?PR?MAIN?OVL1                  -----   -----         -----   -----
  +--> ?PR?INDIRECTFUNC1?OVL1
  +--> ?PR?EXECUTE?OVL1
  +--> ?PR?INDIRECTFUNC2?OVL1

?PR?INDIRECTFUNC1?OVL1         -----   -----         0008H   0002H

?PR?EXECUTE?OVL1               -----   -----         0008H   0004H

?PR?INDIRECTFUNC2?OVL1         -----   -----         0008H   0002H
```

The entry for **?PR?MAIN?OVL1** references **?PR?INDIRECTFUNC1?OVL1**, **?PR?EXECUTE?OVL1**, and **?PR?INDIRECTFUNC2?OVL1**. However, only the function **execute** is called from **main**. The other references are results from using the function pointer **fct**, which is passed to **execute**. The function call to **indirectfunc1** and **indirectfunc2** takes place in **execute,** not in **main** where the function is referenced.

In this situation, the linker/locator cannot locate the actual function calls. Therefore, the BL51 code banking linker/locator incorrectly overlays the local segments of the functions **execute**, **indirectfunc1**, and **indirectfunc2**. This, in turn, overwrites the variable values **i** and **fct**.

You can use **OVERLAY** directive to provide the actual function calls to the linker. For this example, you must remove the references from **`main`** to **`indirectfunc1`** and **`indirectfunc2`**. Do this with **main ~ (indirectfunc1, indirectfunc2)**. Then, add the actual function call from **`execute`** to **`indirectfunc1`** and **`indirectfunc2`** with **executed ! (indirectfunc1, indirectfunc2)**. The following shows the complete linker invocation line for this example.

```
BL51 OVL1.OBJ OVERLAY (main ~    (indirectfunc1, indirectfunc2), &
     execute ! (indirectfunc1, indirectfunc2))
```

The following overlay map shows the corrected references.

```
OVERLAY MAP OF MODULE:   OVL1 (OVL1)

SEGMENT                              BIT-GROUP            DATA-GROUP
  +--> CALLING SEGMENT       START   LENGTH         START   LENGTH
---------------------------------------------------------------------
?C_C51STARTUP                -----   -----          -----   -----
  +--> ?PR?MAIN?OVL1

?PR?MAIN?OVL1                -----   -----          -----   -----
  +--> ?PR?EXECUTE?OVL1

?PR?EXECUTE?OVL1             -----   -----          0008H   0004H
  +--> ?PR?INDIRECTFUNC1?OVL1
  +--> ?PR?INDIRECTFUNC2?OVL1

?PR?INDIRECTFUNC1?OVL1       -----   -----          000CH   0002H

?PR?INDIRECTFUNC2?OVL1       -----   -----          000CH   0002H
```

Example 2: Using an Array with Pointer to Functions

In the following application example, **func1** and **func2** are called indirectly by **main**. The entry points are stored as constant values in the table **functab** and are located in the segment **?CO?modulname**. Therefore, the **?CO?OVL2** segment contains references to **func1** and **func2**.

In reality, however, the calls are executed from the **main** function. But, the BL51 code banking linker/locator assumes that **func1** and **func2** are recursive called, because in **func1** and **func2** constant strings are used. These contants strings are also stored in the segment **?CO?OVL2**. The result is that the BL51 code banking linker/locator reports warnings which indicate recursive calls from the segment **?CO?OVL2** to **func1** and **func2**.

The following listing shows part of the OVL2 program.

```
.
.
.
void func1 (void)  {
  unsigned char i;                                  /* function 1 */

  for (i = 0; i < 10; i++) printf ("THIS IS FUNCTION1\n");
}

void func2 (void)  {                                /* function 2 */
  unsigned char i;

  for (i = 0; i < 10; i++) printf ("THIS IS FUNCTION2\n");
}
```

```
code void (*functab []) () = {func1, func2};       /* function table */

void main (void)  {
  (*functab [P1 & 0x01]) ();
}
.
.
.
```

Although the BL51 code banking linker/locator does not produce erroneous program code in this example, the references should be adjusted to the real calls. The fact is that the functions **`func1`** and **`func2`** are called by the **`main`** function.

The references of the **`?CO?OVL2`** segment to the functions **`func1`** and **`func2`** should be deleted with **?CO?OVL2 ~ (func1, func2)**. Since **`main`** calls **`func1`** and **`func2`** these calls can be defined with **main ! (func1, func2)**. The following shows the complete linker invocation line for the above example.

```
BL51 OVL2.OBJ OVERLAY (?CO?OVL2~(func1, func2), main!(func1, func2))
```

Now, the overlay map shows the corrected references and no warning messages are generated.

```
OVERLAY MAP OF MODULE:   OVL2 (OVL2)

SEGMENT                         BIT-GROUP             DATA-GROUP
  +--> CALLING SEGMENT        START  LENGTH         START  LENGTH
-----------------------------------------------------------------
?C_C51STARTUP                 -----   -----         -----   -----
  +--> ?PR?MAIN?OVL2

?PR?MAIN?OVL2                 -----   -----         -----   -----
  +--> ?C_LIB_CODE
  +--> ?CO?OVL2
  +--> ?PR?FUNC1?OVL2
  +--> ?PR?FUNC2?OVL2

?PR?FUNC1?OVL2                -----   -----         0008H   0001H
  +--> ?CO?OVL2
  +--> ?PR?PRINTF?PRINTF

?PR?PRINTF?PRINTF             -----   -----         0009H   0014H
  +--> ?C_LIB_CODE
  +--> ?PR?PUTCHAR?PUTCHAR

?PR?PUTCHAR?PUTCHAR           -----   ----          001DH   0001H

?PR?FUNC2?OVL2                -----   -----         0008H   0001H
  +--> ?CO?OVL2
  +--> ?PR?PRINTF?PRINTF
```

REGFILE

The **REGFILE** directive allows you to specify the name of the file generated by the BL51 code banking linker/locator that contains register usage flags for each C function in your program.

The information in this file is used by the C51 compiler when generating code for each function invocation. The C51 compiler can use the register usage information generated by the linker to optimize the use of registers when passing values to and returning values from external functions. This directive facilitates global register optimization.

REGFILE must be specified on the command line with a valid file name, for example:

```
BL51 MYPROG.OBJ REGFILE(MYPROG.REG)
```

In this instance, the BL51 code banking linker/locator generates the file **MYPROG.REG** which contains register usage information.

Bank Switching Directives

The BL51 code banking linker/locator manages and allows you to locate program code in up to 32 code banks and one common code area. The common code area is always available to all code banks. These area as well as other aspects of code banking are described below.

Common Code Area

The common code area can be accessed by all banks. This area usually includes routines and constant data that must always be accessible; for example, interrupt and reset vectors, interrupt routines, string constants, bank switching routines, etc. The following code sections must always be located in the common area:

Reset Vectors Reset and interrupt jump entries must remain in the common area

Interrupt Vectors in each case, since the code bank selected by the 8051 program is not known at the time of the CPU reset or interrupt. The BL51 code banking linker/locator, therefore, locates absolute code segments in the common area in each case.

Code Constants

Constant values (strings, tables, etc.) which are defined in the code area must be stored in the common area unless you guarantee that the code bank containing the constant data is selected at the time they are accessed by program code. You can relocate these segments in code banks by means of control statements.

Interrupt Functions

Interrupt functions generated using the C51 compiler must always be located in the common area. Interrupt functions can call functions in other code banks. The BL51 code banking linker/locator produces a warning when an attempt is made to locate a C51 interrupt function in a code bank.

Bank Switch Code

The code required for switching the code banks as well as the associated jump table are located in the common area since these program sections are required by all banks. As a standard procedure, the BL51 code banking linker/locator automatically locates these segments in the common area. You should not attempt to locate these program sections in other bank areas.

Library Functions

Run-time library functions that are invoked by the C51 compiler or the PL/M-51 compiler must be located in the common area. It is possible that the bank switch code may use registers that are used to transfer values to the library functions. Therefore, the BL51 code banking linker/locator always locates program sections of the runtime library in the common area. You should not locate these program sections in other bank areas.

It is difficult to provide a general rule concerning the size of the common area. The size will always depend on the particular software application and hardware constraints.

Typically, a separate ROM will be used for the common code area. If this ROM is not large enough to contain the entire common code, the BL51 code banking linker/locator will duplicate the remainder of the common code area in the beginning of each code bank. You may also specify that the BL51 code banking linker/locator include the entire common area in each code bank and avoid using a separate common area ROM.

Code Bank Areas

The 8051 only provides 16 address lines for accessing code memory. With 16 address lines, only 64 KBytes of code space can be accessed. Code banks are addressed using up to five additional address lines that must originate from 8051 I/O ports or from external hardware devices (latches or PIOs) that are mapped into the XDATA or port memory space. A particular code bank is selected by controlling the state of the additional address lines. Up to 32 banks can be used.

Code banking applications must include the assembly file **L51_BANK.A51** which is located in the **LIB** subdirectory. This source module contains the code that is invoked to switch code banks. You must modify this source file to properly manipulate the bank switching techniques used by your target hardware. Refer to “Bank Switching Configuration” on page 51 for a description of this source file.

Optimum Program Structure with Bank Switching

The BL51 code banking linker/locator automatically generates a jump table for all functions which are stored in the bank area and are called from the common area or from other banks. The BL51 code banking linker/locator only uses bank switching when the program section called actually lies in another memory bank or when it can be called from the common area. This improves performance and prevents bank switching from significantly impacting the performance of your application program. Additionally, the memory and stack requirements for this bank switching technique are considerably smaller than other alternative solutions.

Each bank switch takes approximately 50 processor cycles and requires two additional bytes in the stack area. Bank switches are relatively fast, however, programs should be structured so that bank switches are seldom required to achieve maximum performance. This means that functions that are frequently invoked and functions that are called from multiple code banks should be located in the common code area.

Specifying Code Banks and Common Code Areas

The BL51 code banking linker/locator provides the **BANKAREA**, **BANK*x***, and **COMMON** directives to specify the location and size of the bank switching area, the segments to locate in particular code banks, and the segments to locate in the common area.

BANKAREA

The **BANKAREA** directive allows you to specify the starting and ending address of the area where the code banks will be located. These addresses should reflect the actual address where the code bank ROMs are physically mapped. All segments that are assigned to a bank will be located within this address range unless they are defined differently using the **BANK*x*** directive.

The **BANKAREA** directive must be specified according to the following format,

```
BANKAREA (start, end)
```

where

start is the starting address.

end is the ending address of the code banking area.

Example:

```
BL51 … BANKAREA(8000h, 0FFFFh)
```

This example specifies that the code bank area is 32 KBytes long and is located from 8000h to 0FFFFh.

BANK*x*

When you invoke the BL51 code banking linker/locator for the purpose of generating a code banking application program, you must specify which program code you want located in each code bank. This is accomplished using the **BANK*x*** directive. Program code that is not explicitly located in a code bank will be located in the common area.

The ***x*** in the **BANK*x*** directive should be replaced by the actual bank number which may be a number from 0 to 31. For example, BANK0 for code bank number 0, BANK1 for code bank number 1, and so on.

The **BANK*x*** directive allows you to specify:

- Object and library files to include in the code bank
- Additional segments to include in the code bank.

The **BANK*x*** directive has two distinct forms as shown below.

```
BANKx { filename ⟦(sfname)⟧ ⟦, filename …⟧}
```

or

```
BANKx (⟦saddr ⟦,⟧⟧ ⟦sfname ⟦(addr)⟧ ⟦, sfname …⟧⟧)
```

where

x is the bank number to use and can be a number from 0 to 15.

{ and **}** are used to enclose object files or library files.

(and **)** are used to enclose the names of segments.

filename is the name of an object file or library file.

sfname is the name of a segment or C function.

saddr is the starting address to use for the specified segments.

addr is the starting address for a particular segment.

The first form of the **BANK*x*** directive uses curly braces to enclose the filenames of object and library files. This form of the **BANK*x*** directive may only be specified in the ***inputlist*** portion of the BL51 code banking linker/locator command line.

The second form of the **BANK*x*** directive uses parentheses to enclose the names of program segments. This form of the **BANK*x*** directive may only be specified in the ***directives*** portion of the BL51 code banking linker/locator command line.

Refer to the following section for more information about the **BANK*x*** directive.

COMMON

The **COMMON** directive is identical to the **CODE** directive and performs the same operations. When specifying code banking programs, this directive operates identically to the **BANK*x*** directive and allows you to specify:

- Object and library files to include in the common area
- Additional segments to include in the common area.

The **COMMON** directive has two distinct forms as shown below.

```
COMMON {filename ⟦(sfname)⟧⟦, filename …⟧}
```

or

```
COMMON (⟦saddr⟦,⟧⟧ ⟦sfname ⟦(addr)⟧⟦, sfname …⟧⟧)
```

where

{ and **}** are used to enclose object files or library files.

(and **)** are used to enclose the starting address for the bank and segment names and their starting addresses.

filename is the name of an object file or library file.

sfname is the name of a segment or C function.

saddr is the starting address to use for the specified segments.

addr is the starting address for a segment.

The first form of the **COMMON** directive uses curly braces to enclose the filenames of object and library files. This form of the **COMMON** directive may only be specified in the ***inputlist*** portion of the BL51 code banking linker/locator command line.

The second form of the **COMMON** directive uses parentheses to enclose the names of program segments. This form of the **COMMON** directive may only be specified in the ***directives*** portion of the BL51 code banking linker/locator command line.

Ordering Segments in a Bank

The BL51 code banking linker/locator orders segments within a code bank according to established guidelines.

Segments from object modules and libraries (specified using curly braces) are located starting at the address specified with the **BANKAREA** directive.

Segments (specified using parentheses) are located starting at *saddr* or address 0000h if *saddr* is not specified. Segments may be located at an explicitly specified address.

Segments are located in a code bank in the following order:

1. Segments specified with explicit addresses.
2. Segments specified without explicit addresses.
3. Segments from object and library files.

Example

A typical BL51 code banking linker/locator command line appears as follows:

```
BL51 COMMON{C_ROOT.OBJ}, &
>> BANK0{C_BANK0.OBJ}, &
>> BANK1{C_BANK1.OBJ}, &
>> BANK2{C_BANK2.OBJ} &
>> TO MYPROG.ABS &
>> BANKAREA(8000H,0FFFFH)
```

This example shows how to specify the code bank to use for object modules included in the program linkage.

You may also specify the code bank to use for individual code segments. For example:

```
BL51 COMMON{C_ROOT.OBJ}, &
>> BANK0{C_BANK0.OBJ}, &
>> BANK1{C_BANK1.OBJ} &
>> TO MYPROG2.ABS &
>> BANKAREA(8000H,0FFFFH) &
>> BANK2(8000h, ?PR?FUNC2?C_BANK2)
```

The **BANK2(8000h, ?PR?FUNC2?C_BANK2)** directive specifies that the C function func2 is to be located in bank 2 starting at address 8000h.

You can explicitly specify the starting address for a particular code segment. For example:

```
BL51 COMMON{C_ROOT.OBJ}, &
>> BANK0{C_BANK0.OBJ}, &
>> TO MYPROG3.ABS &
>> BANKAREA(8000H,0FFFFH) &
>> BANK1(8000h, ?PR?FUNC1?C_BANK1, ?PR?FUNC2?C_BANK2(8200h))
```

In this example, the segment **?PR?FUNC1?C_BANK1** is located starting at **8000H** in bank 1. The segment **?PR?FUNC2?C_BANK2** is located at **8200H** in bank 1.

Automatic Bank Selection

The BL51 code banking linker/locator will automatically assign bank numbers in sequence to object files and library files that are specified on the command line enclosed in curly braces. For example:

```
BL51 {C_BANK0.OBJ}, {C_BANK1.OBJ}, {C_BANK2.OBJ}, &
>> C_ROOT.OBJ TO MYPROG4.ABS BANKAREA(8000H,0FFFFH)
```

This example locates code segments from **`C_BANK0.OBJ`** in bank 0, **`C_BANK1.OBJ`** in bank 1, and **`C_BANK2.OBJ`** in bank 2. All other program segments from **`C_ROOT.OBJ`** are located in the common code area.

This is equivalent to the following command line.

```
BL51 COMMON{C_ROOT.OBJ}, &
>> BANK0{C_BANK0.OBJ}, &
>> BANK1{C_BANK1.OBJ}, &
>> BANK2{C_BANK2.OBJ} &
>> TO MYPROG4.ABS &
>> BANKAREA(8000H,0FFFFH)
```

RTX 51 Full and RTX51 Tiny Directives

You must use the BL51 code banking linker/locator when you link programs with the RTX51 and RTX51 Tiny Real-Time Multitasking Operating Systems. The **RTX51 Full** and **RTX51TINY** directives instruct the BL51 code banking linker/locator to resolve references to the RTX51 and RTX51 Tiny libraries respectively.

RTX51

The **RTX51** directive specifies to the BL51 code banking linker/locator that the application should be linked for use with the RTX51 Real-Time Multitasking Operating System. This involves resolving references within your program to RTX51 functions found in the RTX51 library. This directive is specified on the command line as shown in the following example:

```
BL51 RTX_EX1.OBJ RTX51
```

RTX51TINY

The **RTX51TINY** directive specifies to the BL51 code banking linker/locator that the application should be linked for use with the RTX51 Tiny Real-Time Multitasking Operating System. This involves resolving references within your program to RTX51 Tiny functions found in the RTX51 Tiny library. This directive is specified on the command line as shown in the following example:

```
BL51 RTX_EX1.OBJ RTX51TINY
```

Bank Switching Configuration

When you create a code banking application, you must specify the number of code banks your hardware provides as well as how the code banks are switched. This is done by changing constants that are defined in the assembly module **L51_BANK.A51** found in the **\C51\LIB\** subdirectory.

L51_BANK.A51 Constants

The banking method as well as the number of banks and thus the number of address lines used are configured using this source file. **L51_BANK.A51** contains EQU statements at the beginning which are used for the configuration. Following is a listing of these as well as a description of each.

```
;************************ Configuration Section ************************
?B_NBANKS      EQU     32       ; Define max. Number of Banks          *
;                                                                      *
?B_MODE        EQU     0        ; 0 for Bank-Switching via 8051 Port  *
;                               ; 1 for Bank-Switching via XDATA Port *
;                                                                      *
IF  ?B_MODE = 0;                                                       *
;------------------------------------*
; if ?BANK?MODE is 0 define the following values                       *
; For Bank-Switching via 8051 Port define Port Address / Bits         *
?B_PORT         EQU     P1       ; default is P1                       *
?B_FIRSTBIT     EQU     3        ; default is Bit 3                    *
;------------------------------------*
ENDIF;                                                                 *
;                                                                      *
IF  ?B_MODE = 1;                                                       *
;------------------------------------*
; if ?BANK?MODE is 1 define the following values                       *
; For Bank-Switching via XDATA Port define XDATA Port Address / Bits  *
?B_XDATAPORT    EQU     0FFFFH   ; default is XDATA Port Address 0FFFFH*
?B_FIRSTBIT     EQU     0        ; default is Bit 0                    *
;------------------------------------*
ENDIF;                                                                 *
;                                                                      *
;***********************************************************************
```

?B_NBANKS indicates the number of banks to be supported. The number must be between 2 and 32. Only one 8051 address line (port terminal) is used for two banks. Three or four banks require two address lines. Five to eight banks require three address lines. Nine to sixteen banks require four address lines. Seventeen to thirty-two banks require five address lines.

?B_MODE indicates if the bank switching code should use an 8051 port or an XDATA port for the address extension. A value of 0 defines an arbitrary 8051 port for the address extension. A value of 1 determines a XDATA port which is addressed in the external address space of the 8051.

?B_PORT specifies the port address used to select the bank address. If the value 0 is used for **?B_MODE**, **?B_PORT** can be used to specify the address of the internal data port. In this case, the SFR address of an internal data port must be specified. P1 is defined as the default value for port 1.

?B_XDATAPORT specifies the XDATA memory address used to select the bank address. If the value 1 is used for **?B_MODE**, **?B_XDATAPORT** defines the address of an external data port. In this case, an arbitrary XDATA address can be specified (address range 0H to 0FFFFH) under which a port can be addressed in the XDATA area. 0FFFFH is defined as the default value. If either Intel PL/M-51 or the A51 Assembler is used, the memory locations **?B_CURRENTBANK** and **?B_XDATAPORT** must be initialized with the value 0 at the start of the program.

?B_FIRSTBIT indicates which bit of the defined port is to be assigned first. The value **?B_FIRSTBIT EQU 3** (defined as the default when **?B_MODE** is 0) indicates that P1.3 is to be used as the first port terminal for the address extension. If, for example, two port terminals are used for the extension, P1.3 and P1.4 are used in this case. The remaining lines of the 8051 port can be used for other purposes. If the value 1 is selected for **?B_MODE**, the remaining bits of the XDATA port cannot be used for other purposes.

The A51 assembler is required to assemble **L51_BANK.A51**. The object file **L51_BANK.OBJ** is automatically linked to the application if the standard default setting (**DEFAULTLIBRARY**) is used by the BL51 code banking linker/locator, and when a high-level language library was added. Otherwise, **L51_BANK.OBJ** must be specified as a file in the input list for the BL51 code banking linker/locator.

Public Symbols in L51_BANK.A51

Additional PUBLIC Symbols are provided in **L51_BANK.A51** for your convenience. They are described below.

?B_CURRENTBANK is a memory location in the DATA or SFR memory which contains the currently selected memory bank. This memory location can be read for debugging. A modification of the memory location, however, does not cause a bank switching in most cases. Note that the bits are only valid which are required in this memory location based on setting **?B_NBANKS** and **?B_FIRSTBIT.** For this reason, the bits which are not required must be masked out by means of a corresponding mask.

_SWITCHBANK is a C51 compatible function which allows the bank address to be selected by the user program. This function can be used for bank switching if the constant memory is too small. This C function can be accessed as follows:

```
extern void switchbank (
  unsigned char bank_number);
.
.
.
switchbank (0);
```

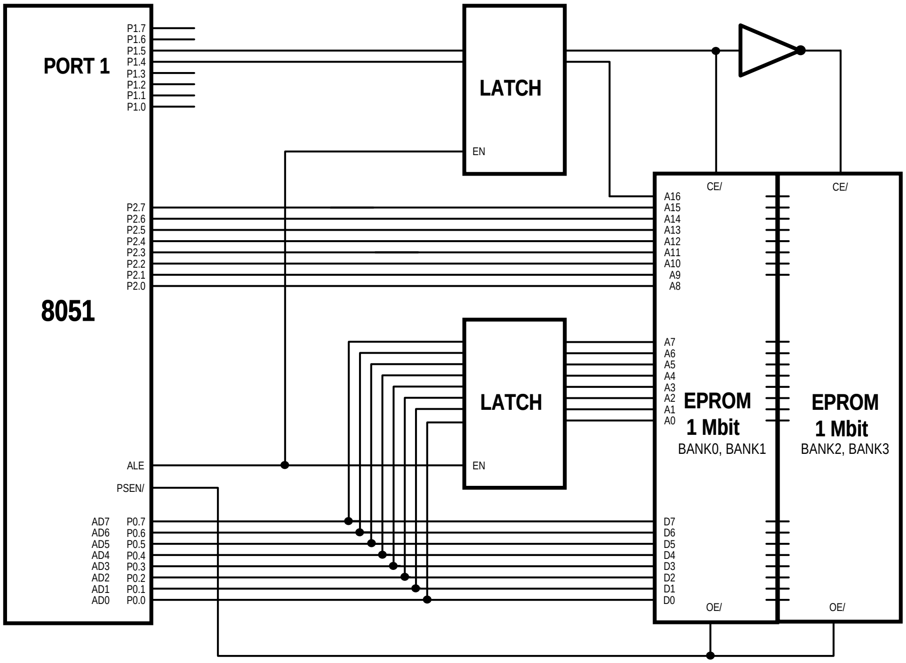
NOTE
The function ***switchbank*** *may only be invoked from the common area.*

Configuration Examples

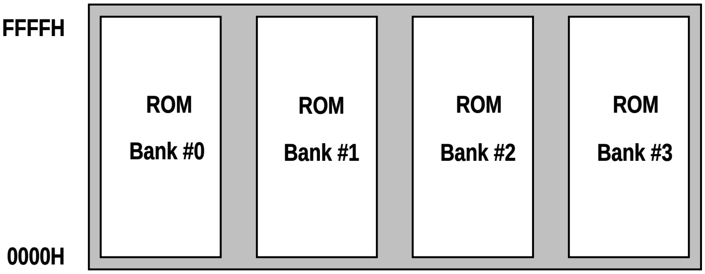
The following examples demonstrate how to configure **L51_BANK.A51** for several different hardware scenarios.

Banking With Four 64 KByte Banks

This example demonstrates the configuration required to bank switch using two 1 Mbit EPROMs. The following figure illustrates the hardware schematic.



The following figure illustrates the memory map for this example.



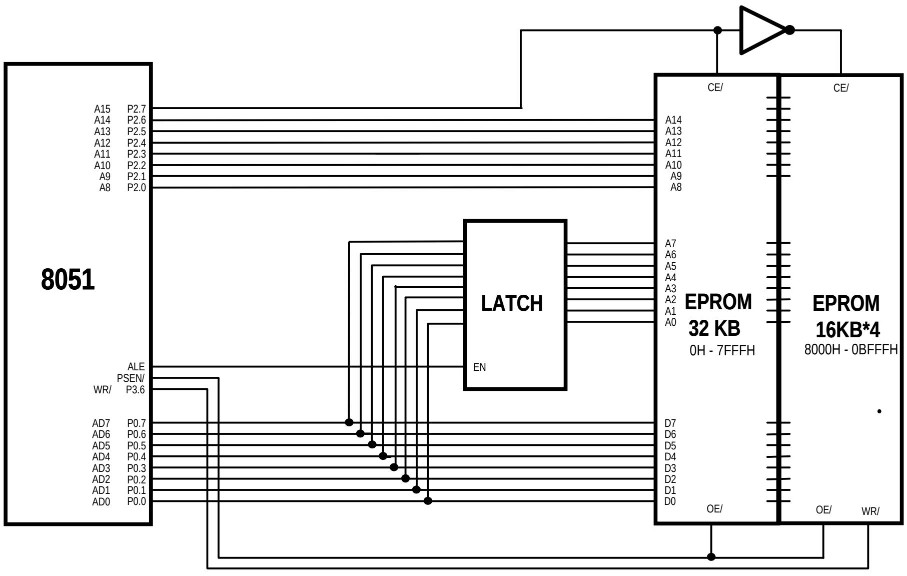
Two 128KB EPROMs are used in this hardware configuration. The bank switching can be implemented by using two bank select address lines (Port 1.4 and Port 1.5). **L51_BANK.A51** can be configured as follows for this hardware configuration.

```
?N_BANKS       EQU  4       ; Four banks are required.
?B_MODE        EQU  0       ; 8051 port is used.
?B_PORT        EQU  090H    ; Port 1 as address line.
?B_FIRSTBIT    EQU  4       ; P1.4 is the 1st address line.
```

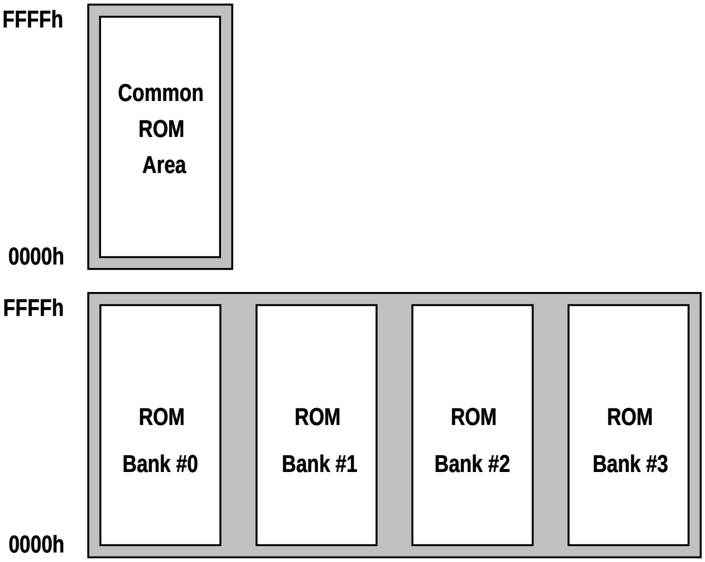
The BL51 code banking linker/locator automatically places copies of the code and data in the common area into each bank so that the contents of all EPROM banks are identical in the address range of the common area. The **BANKAREA** directive should not be specified since the default setting already defines address space 0000h to 0FFFFh as the bank area.

Banking With a 32 KByte Common Area and Four 16 KByte Banks

This example demonstrates the configuration required to bank switch using four 16 KByte EPROMs. The application uses a EPROM with on-chip bank switching logic. The following figure illustrates the hardware schematic.



The following figure illustrates the memory map for this example.



The hardware consists of four memory banks with 16 KBytes each and a common area consisting of 32 KBytes. The bank switching will be implemented via XDATA address 8000h. **L51_BANK.A51** can be configured as follows for this hardware configuration.

```
?N_BANKS      EQU   4         ; Four banks are required.
?B_MODE       EQU   1         ; XDATA port is used.
?B_XDATAPORT  EQU   08000H    ; Port address is 8000H.
?B_FIRSTBIT   EQU   0         ; Bit 0 is the 1st address line.
```

In the BL51 code banking linker/locator command line, the address space from 08000h to 0BFFFh should be defined as the bank area using the **BANKAREA** directive.

BL51 Directive Reference

This section lists all BL51 directives in alphabetical order.

Many of the BL51 code banking linker/locator directives allow you to specify optional arguments and parameters in parentheses immediately following the directive. The following table lists the types of arguments that are allowed with certain directives.

Argument	Description
address	A 16-bit value representing a code or data memory location.
filename	The name of a DOS file which must adhere to the following format: ⟦*drive* :⟧ ⟦*directory* \⟧ *file* ⟦. *ext*⟧ *where* *drive* is a valid disk drive letter (A-Z). *directory* is the name of a valid MS-DOS directory path. *file* is the file name. *ext* is the file extension.
modname	A module name which may be up to 40 characters long and must adhere to the following format: { A—Z | ? | _ | @ } ⟦{ A—Z | 0—9 | ? | _ | @ }⟧
segname	A segment name which may be up to 40 characters long and must adhere to the following format: { A—Z | ? | _ | @ } ⟦{ A—Z | 0—9 | ? | _ | @ }⟧
sfname	A segment or function name which may be up to 40 characters long and must adhere to the following format: { A—Z | ? | _ | @ } ⟦{ A—Z | 0—9 | ? | _ | @ }⟧
value	A 16-bit value, for example, 1011B, 2048D, or 0D5FFh.

BANKAREA

Abbreviation: **BA**

Arguments: **BANKAREA (*start_address*, *end_address*)**

Default: None

Description: Use the **BANKAREA** directive to specify the starting and ending address of the area where the code banks will be located. The addresses specified should reflect the actual address where the code bank ROMs are physically mapped. All segments that are assigned to a bank will be located within this address range unless they are defined differently using the **BANK*x*** directive. Refer to “Bank Switching Directives” on page 43 for more information about the code banking directives.

NOTE
This control is not available in L51.

See Also: **BANK*x*, COMMON**

Example:

```
BL51 COMMON{C_ROOT.OBJ}, &
>> BANK0{C_BANK0.OBJ}, &
>> BANK0{C_BANK0.OBJ}, &
>> BANK1{C_BANK1.OBJ}, &
>> BANK2{C_BANK2.OBJ} &
>> TO MYPROG.ABS &
>> BANKAREA(8000H,0FFFFH)
```

BANK*x*

Abbreviation: **B0, B1, B2, … B30, B31**

Arguments: **BANK*x* {*filename* ⟦(*sfname*)⟧⟦, *filename* …⟧}**

BANK*x* (⟦*start_address* ⟦,⟧⟧⟦*sfname* ⟦(*address*)⟧
⟦, *sfname*…⟧⟧)

Default: None

Description: Use the **BANK*x*** directive to specify object modules, library files, and segments to include in a specific code bank. The ***x*** in the **BANK*x*** directive should be replaced by the actual bank number which may be a number from 0 to 31. Refer to “Bank Switching Directives” on page 43 for more information about the code banking directives.

NOTE
This control is not available in L51.

See Also: **BANKAREA, COMMON**

Example:

```
BL51 COMMON{C_ROOT.OBJ}, &
>> BANK0{C_BANK0.OBJ}, &
>> BANK1{C_BANK1.OBJ}, &
>> BANK2{C_BANK2.OBJ} &
>> TO MYPROG.ABS &
>> BANKAREA(8000H,0FFFFH)
```

BIT

Abbreviation: **BI**

Arguments: **BIT ({ *address* | *segname* ⟦(*address*)⟧⟦, …⟧})**

Description: The **BIT** directive allows you to specify:

- The starting address for segments placed in the bit-addressable internal data space
- The order of segments within the bit-addressable internal data space
- The absolute memory location of segments in the bit-addressable internal data space.

Addresses that you specify with the BIT directive are bit addresses. In the 8051, bit addresses 00h through 7Fh reference bits in internal data memory bytes from byte address 20h to 2Fh (16 bytes of 8 bits each, 16 × 8 = 128 = 80h). Bit addresses that are evenly divisible by 8 reference the low-order bit for its corresponding byte and are also considered to be aligned on a byte border. A DATA segment that is bit-addressable can be located with the BIT directive; however, the bit address specified must lie on a byte boundary. The bit address must be evenly divisible by 8. Refer to “Segment Size and Location Directives” on page 25 for more information about this directive.

See Also: **CODE, DATA, IDATA, XDATA**

Example:

```
BL51 MYPROG.OBJ BIT(20h.2)

BL51 MYPROG.OBJ,A.OBJ,B.OBJ,C.OBJ BIT(?DT?A,?DT?B,?DT?C)

BL51 MYPROG.OBJ,A.OBJ,B.OBJ BIT(?DT?A(28h),?DT?B(30h))
```

CODE

Abbreviation: **CO**

Arguments: **CODE ({ *address* | *segname* ⟦(*address*)⟧ ⟦, …⟧})**

Description: The **CODE** directive allows you to specify:

- The starting address for segments placed in the code memory space
- The order of segments within the code memory space
- The absolute memory location of segments in the code memory space.

Refer to “Segment Size and Location Directives” on page 25 for more information about this directive.

See Also: **BIT, DATA, IDATA, XDATA**

Example:

```
BL51 MYPROG.OBJ CODE(4000h)

BL51 MYPROG.OBJ CODE(?PR?FUNC1?MYPROG,?PR?FUNC2?MYPROG)

BL51 MYPROG.OBJ &
>> CODE(?PR?FUNC1?MYPROG(1000h), &
>> ?PR?FUNC2?MYPROG(2000h))
```

COMMON

Abbreviation: **CO**

Arguments: **COMMON {*filename* ⟦(*sfname*)⟧⟦, *filename* …⟧}**

COMMON (⟦*saddr* ⟦,⟧⟧⟦*sfname*⟦(*addr*)⟧⟦, *sfname* …⟧⟧)

Default: None

Description: The **COMMON** directive allows you to specify object modules, library files, and segments to include in the common code area when using bank switching. Refer to “Bank Switching Directives” on page 43 for more information about the code banking directives.

NOTE
This control is not available in L51.

See Also: **BANK*x*, BANKAREA**

Example:

```
BL51 COMMON{C_ROOT.OBJ}, &
>> BANK0{C_BANK0.OBJ}, &
>> BANK1{C_BANK1.OBJ}, &
>> BANK2{C_BANK2.OBJ} &
>> TO MYPROG.ABS &
>> BANKAREA(8000H,0FFFFH)
```

DATA

Abbreviation: **DA**

Arguments: **DATA ({*address* | *segname* ⟦(*address*)⟧⟦, …⟧})**

Description: The **DATA** directive allows you to specify:

- The starting address for segments placed in the directly-addressable internal data space
- The order of segments within the directly-addressable internal data space
- The absolute memory location of segments in the directly-addressable internal data space.

Refer to “Segment Size and Location Directives” on page 25 for more information about this directive.

See Also: **BIT, CODE, IDATA, XDATA**

Example:

```
BL51 MYPROG.OBJ DATA(30h)
BL51 MYPROG.OBJ,A.OBJ,B.OBJ,C.OBJ
DATA(?DT?A,?DT?B,?DT?C)
BL51 MYPROG.OBJ,A.OBJ,B.OBJ DATA(?DT?A(28h),?DT?B(30h))
```

IDATA

Abbreviation: **ID**

Arguments: **IDATA ({*address* | *segname* ⟦(*address*)⟧⟦, …⟧})**

Description: The **IDATA** directive allows you to specify:

- The starting address for segments placed in the indirectly-addressable internal data space
- The order of segments within the indirectly-addressable internal data space
- The absolute memory location of segments in the indirectly-addressable internal data space.

Refer to “Segment Size and Location Directives” on page 25 for more information about this directive.

See Also: **BIT, CODE, DATA, XDATA**

Example:

```
BL51 MYPROG.OBJ IDATA(40h)

BL51 MYPROG.OBJ,A.OBJ,B.OBJ,C.OBJ &
>> IDATA(?ID?A,?ID?B,?ID?C)

BL51 MYPROG.OBJ,A.OBJ,B.OBJ &
>> IDATA(?ID?A(30h),?ID?B(40h))
```

IXREF

Abbreviation: **IX**

Arguments: **IXREF** ⟦**(NOGENERATED, NOLIBRARIES)**⟧

Default: No cross reference is generated.

Description: The **IXREF** directive instructs the BL51 code banking linker/locator to include a cross reference report in the listing file. A cross reference report lists symbols, the area of memory in which they are located (for example, **CODE**, **XDATA**, **DATA**, and **BIT**), and the source modules in which they are accessed.

The option **NOGENERATED** suppresses symbols starting with ‘?’. These question mark symbols are normally produced by the compiler for calling specific C functions or passing parameters.

The option **NOLIBRARIES** suppresses those symbols which are defined in a library file.

Example:

```
BL51 myfile.obj IXREF

BL51 myfile.obj IXREF (NOGENERATED)

BL51 myfile.obj IXREF(NOLIBRARIES, NOGENERATED)
```

NAME

Abbreviation: **NA**

Arguments: **NAME (*modname*)**

Default: The basename of the first object file in the input list is used.

Description: Use the **NAME** directive to specify a module name for the absolute object module that the BL51 code banking linker/locator generates. The **NAME** directive may be accompanied by the module name (in parentheses) that you want to assign. Refer to “Output File Directives” on page 22 for more information about this directive.

Example:

```
BL51 MYPROG.OBJ TO MYPROG.ABS NAME(BIGPROG)
```

NOAMAKE

Abbreviation: None

Arguments: None

Default: **AMAKE**

Description: The **NOAMAKE** directive allows you to direct the linker to exclude **AMAKE** information from the generated absolute object file. By default, the BL51 code banking linker/locator generates object modules that include records containing time and date information for the source files and include files used to build specific object modules.

Example:

```
BL51 MYPROG.OBJ TO MYPROG.ABS NOAMAKE
```

NODEBUGLINES

Abbreviation: **NODL**

Arguments: None

Default: **DEBUGLINES**

Description: The **NODEBUGLINES** directive directs the BL51 code banking linker/locator to exclude line number information from the object file. Refer to “Output File Directives” on page 22 for more information about this directive.

See Also: **DEBUGLINES**

Example:

```
BL51 MYPROG.OBJ NODEBUGLINES
```

NODEBUGPUBLICS

Abbreviation: **NODP**

Arguments: None

Default: **DEBUGPUBLICS**

Description: The **NODEBUGPUBLICS** directive directs the BL51 code banking linker/locator to exclude public symbol information from the object file. Refer to “Output File Directives” on page 22 for more information about this directive.

See Also: **DEBUGPUBLICS**

Example:

```
BL51 MYPROG.OBJ NODEBUGPUBLICS
```

NODEBUGSYMBOLS

Abbreviation: **NODS**

Arguments: None

Default: **DEBUGSYMBOLS**

Description: The **NODEBUGSYMBOLS** directive directs the BL51 code banking linker/locator to exclude local symbol information from the object file. Refer to “Output File Directives” on page 22 for more information about this directive.

See Also: **DEBUGSYMBOLS**

Example:

```
BL51 MYPROG.OBJ NODEBUGSYMBOLS
```

NODEFAULTLIBRARY

Abbreviation: **NLIB**

Arguments: None

Default: Library files are searched to resolve external references.

Description: Use the **NODEFAULTLIBRARY** directive to prevent the BL51 code banking linker/locator from including modules from the run-time libraries.

Example:

```
BL51 MYPROG.OBJ NODEFAULTLIBRARY
```

NOLINES

Abbreviation: **NOLI**

Arguments: None

Default: **LINES**

Description: The **NOLINES** directive prevents the BL51 code banking linker/locator from including line number information in the listing file. Refer to “Listing File Directives” on page 17 for more information about this directive.

See Also: **LINES**

Example:

```
BL51 MYPROG.OBJ NOLINES
```

NOMAP

Abbreviation: **NOMA**

Arguments: None

Default: **MAP**

Description: The **NOMAP** directive prevents the BL51 code banking linker/locator from including the memory map in the listing file. Refer to “Listing File Directives” on page 17 for more information about this directive.

See Also: **MAP**

Example:

```
BL51 MYPROG.OBJ NOMAP
```

NOPUBLICS

Abbreviation: **NOPU**

Arguments: None

Default: **PUBLICS**

Description: The **NOPUBLICS** directive instructs the BL51 code banking linker/locator to exclude public symbols from the listing file. Refer to “Listing File Directives” on page 17 for more information about this directive.

See Also: **PUBLICS**

Example:

```
BL51 MYPROG.OBJ NOPUBLICS
```

NOSYMBOLS

Abbreviation: **NOSY**

Arguments: None

Default: **SYMBOLS**

Description: The NOSYMBOLS directive instructs the BL51 code banking linker/locator to exclude local symbols from the listing file. Refer to “Listing File Directives” on page 17 for more information about this directive.

See Also: **SYMBOLS**

Example:

```
BL51 MYPROG.OBJ NOSYMBOLS
```

OVERLAY / NOOVERLAY

Abbreviation: **OL / NOOL**

Arguments: **OVERLAY (*sfname* { ! | ~ } *sfname* ⟦, …⟧)**

OVERLAY (*sfname* { ! | ~ } (*sfname*, *sfname* ⟦, …⟧)⟦, …⟧)

OVERLAY (*sfname* ! *)

OVERLAY (* ! *sfname*)

Default: **OVERLAY**

Description: The **OVERLAY** directive allows you to control the inter-segment references that the BL51 code banking linker/locator uses in its overlay analysis. The **OVERLAY** directive may be specified a number of times in the command line for each reference. The general format of the overlay parameter may be any one of the following:

Directive Specification	Description
OVERLAY (* ! *sfname*)	Used to add new roots for *sfname*.
OVERLAY (*sfname* ! *)	Used to exclude *sfname* from the overlay analysis and process it in a normal fashion. This has no influence on the overlay evaluation of other segments.
OVERLAY (*sfname* ! *sfname1*) **OVERLAY** (*sfname* ! (*sfname1*, *sfname2*))	Used to add references to segments or functions.
OVERLAY (*sfname* ~ *sfname1*) **OVERLAY** (*sfname* ~ (*sfname1*, *sfname2*))	Used to delete or remove references between segments or functions.

Use the **NOOVERLAY** directive to disable overlay analysis and implementation. When this directive is specified on the command line, the BL51 code banking linker/locator does not overlay variables and function argument data space.

Examples:

```
BL51 MYPROG.OBJ OVERLAY(*! (TASK1, TASK2))

BL51 SAMPLE.OBJ OVERLAY (FUNC1 ! *)

BL51 CMODUL1.OBJ OVERLAY (FUNC1 ! (FUNC2, FUNC3))

BL51 MAINMOD.OBJ, TEXTOUT.OBJ &
>> OVERLAY (FUNC1 ~ ?CO?MAINMOD, FUNC2 ~ ?CO?MAINMOD)

BL51 MYPROG.OBJ NOOVERLAY
```

PAGELENGTH

Abbreviation: **PL**

Arguments: **PAGELENGTH (*value*)**

Default: **PAGELENGTH (68)**

Description: The **PAGELENGTH** directive sets the maximum number of lines per page for the listing file. The minimum page length is 10 lines. Refer to “Listing File Directives” on page 17 for more information about this directive.

See Also: **PAGEWIDTH**

Example:

```
BL51 PROG.OBJ TO PROG.ABS PAGELENGTH(50) PAGEWIDTH(100)
```

PAGEWIDTH

Abbreviation: **PW**

Arguments: **PAGEWIDTH (*value*)**

Default: **PAGEWIDTH (78)**

Description: The **PAGEWIDTH** directive defines the maximum width of lines in the listing file. The page width may be set to a number in the 72 to 132 range. Refer to “Listing File Directives” on page 17 for more information about this directive.

See Also: **PAGELENGTH, PRINT**

Example:

```
BL51 PROG.OBJ TO PROG.ABS PAGELENGTH(50) PAGEWIDTH(100)
```

PDATA

Abbreviation: None

Arguments: **PDATA (*address*)**

Description: The **PDATA** directive allows you to specify the starting address in external data space for **PDATA** segments. You must enter the starting address immediately following the **PDATA** directive on the command line. The address must be enclosed in parentheses. Refer to “Segment Size and Location Directives” on page 25 for more information about this directive.

See Also: **XDATA**

Example:

```
BL51 MYPROG.OBJ PDATA(8000h)
```

PRECEDE

Abbreviation: **PC**

Arguments: **PRECEDE (*segname* ⟦(*address*)⟧⟦, …⟧)**

Description: The **PRECEDE** directive allows you to specify segments that lie in the internal data memory that should precede all other segments in that memory space. Segments that you specify with this directive are located after the BL51 code banking linker/locator has located register banks and any absolute **BIT**, **DATA**, and **IDATA** segments, but before any other segments in the internal data memory. Refer to “Segment Size and Location Directives” on page 25 for more information about this directive.

See Also: **STACK**

Example:

```
BL51 MYPROG.OBJ,A.OBJ,B.OBJ PRECEDE(?DT?A,?DT?B)

BL51 MYPROG.OBJ,A.OBJ,B.OBJ &
>> PRECEDE(?DT?A(09h),?DT?B(13h))
```

PRINT

Abbreviation: **PR**

Arguments: **PRINT (*filename*)**

Default: The listing file is generated using the basename of the output file.

Description: The **PRINT** directive allows you to specify the name of the listing file that is generated by the BL51 code banking linker/locator. The name must be enclosed in parentheses immediately following the **PRINT** directive on the command line. Refer to “Listing File Directives” on page 17 for more information about this directive.

See Also: **PAGELENGTH, PAGEWIDTH**

Example:

```
BL51 MYPROG.OBJ TO MYPROG.ABS PRINT(OUTPUT.MAP)
```

RAMSIZE

Abbreviation: **RS**

Arguments: **RAMSIZE (*value*)**

Default: **RAMSIZE (128)**

Description: The **RAMSIZE** directive allows you to specify the number of bytes of internal data memory that are available in your target 8051 derivative. The number of bytes must be a number between 64 and 256. This number must be enclosed in parentheses. Refer to “Segment Size and Location Directives” on page 25 for more information about this directive.

Example:

```
BL51 MYPROG.OBJ RAMSIZE(256)
```

REGFILE

Abbreviation: **RF**

Arguments: **REGFILE (*filename*)**

Description: The **REGFILE** directive allows you to specify the name of the register usage file generated by the BL51 code banking linker/locator. The information in this file is used by the C51 compiler when generating code for each function invocation. The C51 compiler uses the register usage information generated by the linker to optimize the use of registers when passing values to and returning values from external functions. This directive facilitates global register optimization.

Example:

```
BL51 MYPROG.OBJ,A.OBJ,B.OBJ REGFILE(PROG.REG)
```

RTX51

Abbreviation: None

Arguments: None

Default: None

Description: The **RTX51** directive specifies to the BL51 code banking linker/locator that the application should be linked for use with the RTX51 Full Real-Time Multitasking Operating System. This involves resolving references within your program to RTX51 Full functions found in the RTX51 Full library.

NOTE
This control is not available in L51.

See Also: **RTX51TINY**

Example:

```
BL51 RTX_EX1.OBJ RTX51
```

RTX51TINY

Abbreviation: None

Arguments: None

Default: None

Description: The **RTX51TINY** directive specifies to the BL51 code banking linker/locator that the application should be linked for use with the RTX51 Tiny Real-Time Multitasking Operating System. This involves resolving references within your program to RTX51 Tiny functions found in the RTX51 Tiny library.

NOTE
This control is not available in L51.

See Also: **RTX51**

Example:

```
BL51 RTX_EX1.OBJ RTX51TINY
```

STACK

Abbreviation: **ST**

Arguments: **STACK (*segname* ⟦(*address*)⟧⟦, …⟧)**

Description: The **STACK** directive allows you to specify the segments which are to be located in the uppermost IDATA memory space in internal data memory. The segments you specify with this directive will follow all other segments in the internal data memory space. Refer to “Segment Size and Location Directives” on page 25 for more information about this directive.

See Also: **PRECEDE**

Example:

```
BL51 MYPROG.OBJ,A.OBJ,B.OBJ STACK(?DT?A,?DT?B)

BL51 MYPROG.OBJ,A.OBJ,B.OBJ STACK(?DT?A(69h),?DT?B(73h))
```

XDATA

Abbreviation: **XD**

Arguments: **XDATA ({*address* | *segname* ⟦(*address*)⟧⟦, …⟧})**

Description: The **XDATA** directive allows you to specify:

- The starting address for segments placed in the external data space
- The order of segments within the external data space
- The absolute memory location of segments in the external data space.

Refer to “Segment Size and Location Directives” on page 25 for more information about this directive.

See Also: **BIT, CODE, DATA, IDATA, PDATA**

Example:

```
BL51 MYPROG.OBJ XDATA(1000h)

BL51 MYPROG.OBJ,A.OBJ,B.OBJ,C.OBJ &
>> XDATA(?XD?A,?XD?B,?XD?C)

BL51 MYPROG.OBJ,A.OBJ,B.OBJ &
>> XDATA(?XD?A(100h),?XD?B(200h))
```

BL51 Error Messages

The BL51 code banking linker/locator generates error messages that describe warnings, non-fatal errors, fatal errors, and exceptions.

Fatal errors immediately abort the BL51 code banking linker/locator operation.

Errors and warnings do not abort the BL51 code banking linker/locator operation; however, they may result in an output module that cannot be used. Errors and warnings generate messages that may or may not have been intended by the user. The listing file can be very useful in such an instance. Error and warning messages are displayed in the listing file as well as on the screen.

This section displays all the BL51 code banking linker/locator error messages, their causes, and any recovery actions.

Warnings

Warning	Warning Message and Description
1	**UNRESOLVED EXTERNAL SYMBOL** **SYMBOL: external-name** **MODULE: filename (modulename)** The specified external symbol, requested in the specified module, has no corresponding PUBLIC symbol in any of the input files.
2	**REFERENCE MADE TO UNRESOLVED EXTERNAL** **SYMBOL: external-name** **MODULE: filename (modulename)** **ADDRESS: code-address** The specified unresolved external symbol is referenced at the specified code address.
3	**ASSIGNED ADDRESS NOT COMPATIBLE WITH ALIGNMENT** **SEGMENT: segment–name** The address specified for the segment is not compatible with the alignment of the segment declaration.
4	**DATA SPACE MEMORY OVERLAP** **FROM: byte.bit address** **TO: byte.bit address** The specified area of the on-chip data RAM is occupied by more than one segment.

Warning	Warning Message and Description
5	**CODE SPACE MEMORY OVERLAP** **FROM: byte address** **TO: byte address** The specified area of the code memory is occupied by more than one segment.
6	**XDATA SPACE MEMORY OVERLAP** **FROM: byte address** **TO: byte address** The specified area of the external data memory is occupied by more than one segment.
7	**MODULE NAME NOT UNIQUE** **MODULE: filename (modulename)** The specified module name is used for more than one module. The specified module name is not processed.
8	**MODULE NAME EXPLICITLY REQUESTED FROM ANOTHER FILE** **MODULE: filename (modulename)** The specified module name is requested in the invocation line of another file that has not yet been processed. The specified module name is not processed.
9	**EMPTY ABSOLUTE SEGMENT** **MODULE: filename (modulename)** The specified module contains an empty absolute segment. This segment is not located and may be overlapped with another segment without any additional message.
10	**CANNOT DETERMINE ROOT SEGMENT** The Linker/Locator has recognized the C51 compiler or PL/M-51 input files and tries to process a flow analysis. However it is impossible to determine the root segment. This error occurs if the main program is called by an assembly module. In this case the available references (calls) must be modified with the OVERLAY directive.
11	**CANNOT FIND SEGMENT OR FUNCTION NAME** **NAME: overlay-control-name** A segment or function name defined in the OVERLAY directive cannot be found in the object modules.
12	**NO REFERENCE BETWEEN SEGMENTS** **SEGMENT1: segment-name** **SEGMENT2: segment-name** An attempt was made to delete a reference or call between two non-existent functions or segments, with the OVERLAY directive.

Warning	Warning Message and Description
13	**RECURSIVE CALL TO SEGMENT** **SEGMENT: segment-name** **CALLER: segment-name** The specified segment is called recursively from CALLER specified segments. Recursive calls are not allowed in C51 and PL/M-51 programs.
14	**INCOMPATIBLE MEMORY MODEL** **MODULE: filename (modulename)** **MODEL: memory model** The specified module is not compiled in the same memory model as the former compiled modules. The memory model of the improper module is showed by MODEL.
15	**MULTIPLE CALL TO SEGMENT** **SEGMENT: segment-name** **CALLER1: segment-name** **CALLER2: segment-name** The specified segment is called from two levels, CALLER1, and CALLER2; e.g., main and interrupt program. This has the same effect as a recursive call and may thus lead to the overwriting of parameters or data.
16	**UNCALLED SEGMENT, IGNORED FOR OVERLAY PROCESS** **SEGMENT: segment-name** This warning occurs when functions which were not previously called are contained in a program (e.g., for test purposes). The function specified is excluded from the overlay process in this case. It is possible that the program then occupies more memory as during a call of the specified segment.
17	**INTERRUPT FUNCTION IN BANKS NOT ALLOWED** **SYMBOL: function-name** **SPACE: code-bank** The specified C function is an interrupt function (a C51 function) that was specified to be located in a code bank. Interrupt functions cannot be located in a code bank.

Non-Fatal Errors

Error	Error Message and Description
101	**SEGMENT COMBINATION ERROR** **SEGMENT: segment-name** **MODULE: filename (modulename)** The attributes of the specified partial segment in the specified module cannot be combined with the attributes of the previous defined partial segments of the same name. The partial segment is ignored.
102	**EXTERNAL ATTRIBUTE MISMATCH** **SYMBOL: external-name** **MODULE: filename (modulename)** The attributes of the specified external symbol in the specified module do not match the attributes of the previously defined external symbols. The specified symbol is ignored.
103	**EXTERNAL ATTRIBUTE DO NOT MATCH PUBLIC** **SYMBOL: public-name** **MODULE: filename (modulename)** The attributes of the specified public symbols in the specified module do not match the attributes of the previous defined external symbols. The specified symbol is ignored.
104	**MULTIPLE PUBLIC DEFINITIONS** **SYMBOL: public-name** **MODULE: filename (modulename)** The specified public symbol in the specified module has already been defined in a previously processed file.
105	**PUBLIC REFERS TO IGNORED SEGMENT** **SYMBOL: public-name** **SEGMENT: segment-name** The specified public symbol is defined in the specified segment. It cannot be processed on account of an error. The public symbol is therefore ignored.
106	**SEGMENT OVERFLOW** **SEGMENT: segment-name** The specified segment is longer than 64 KByte and cannot be processed.
107	**ADDRESS SPACE OVERFLOW** **SPACE: space-name** **SEGMENT: segment-name** The specified segment cannot be located at the specified address space. The segment is ignored.

Error	Error Message and Description
108	**SEGMENT IN LOCATING CONTROL CANNOT BE ALLOCATED** **SEGMENT: segment-name** The specified segment in the invocation line cannot be processed on account of its attributes.
109	**EMPTY RELOCATABLE SEGMENT** **SEGMENT: segment-name** The specified segment after combination has a zero size. The specified segment is ignored.
110	**CANNOT FIND SEGMENT** **SEGMENT: segment-name** The specified segment is contained in the invocation line but cannot be found in an input module. The specified segment is ignored.
111	**SPECIFIED BIT ADDRESS NOT ON BYTE BOUNDARY** **SEGMENT: segment-name** The specified segment contained in the BIT directive is a DATA segment. The specified BIT address however is not on a byte boundary. The segment is ignored.
112	**SEGMENT TYPE NOT LEGAL FOR COMMAND** **SEGMENT: segment-name** The specified segment cannot be processed because it does not have a legal type.
114	**SEGMENT DOES NOT FIT** **SPACE: space-name** **SEGMENT: segment-name** **BASE: base-address** **LENGTH: segment-length** The specified segment cannot be located at the base address in the specified address space because of its length. The segment is ignored.
115	**INPAGE SEGMENT IS GREATER THAN 256 BYTES** **SEGMENT: segment-name** The specified segment with the attributes PAGE or INPAGE is greater than 256 bytes. The segment is ignored.
116	**INBLOCK SEGMENT IS GREATER THAN 2048 BYTES** **SEGMENT: segment-name** The specified segment with the attribute INBLOCK is greater than 2048 bytes. The segment is ignored.

Error	Error Message and Description
117	**BIT ADDRESSABLE SEGMENT IS GREATER THAN 16 BYTES** **SEGMENT: segment-name** The specified bit or data segment that was declared with the BITADDRESSABLE attribute is larger than 16 bytes. The segment is not ignored.
118	**REFERENCE MADE TO ERRONEOUS EXTERNAL** **SYMBOL: symbol-name** **MODULE: filename (modulename)** **ADDRESS: code-address** The specified external symbol that was erroneously processed, is referenced in the specified code address.
119	**REFERENCE MADE TO ERRONEOUS SEGMENT** **SEGMENT: symbol-name** **MODULE: filename (modulename)** **ADDRESS: code-address** The specified segment processed with an error, is referenced in the specified code address.
120	**CONTENT BELONGS TO ERRONEOUS SEGMENT** **SEGMENT: segment-name** **MODULE: filename (modulename)** A specified segment that was erroneously processed, is referenced at a specific code address. The segment contents are not available.
121	**IMPROPER FIXUP** **MODULE: filename (modulename)** **SEGMENT: segment-name** **OFFSET: segment-address** After evaluation of absolute fixups, an address is not accessible. The improper address along with the specific module name, partial segment, and segment address are displayed. The fixup command is not processed.
122	**CANNOT FIND MODULE** **MODULE: filename (modulename)** The module specified in the invocation line cannot be found in the input file.
123	**ABSOLUTE DATA/IDATA SEGMENT DOES NOT FIT** **MODULE: filename (modulename)** **FROM: byte address** **TO: byte address** An absolute DATA or IDATA segment contained in the specified module is not permissible due to a conflict with the value specified with the RAMSIZE directive. The absolute segment cannot be located in the area which was output.

Error	Error Message and Description
124	**BANK SWITCH MODULE INCORRECT** This error message is issued when the bank switch module file (L51_BANK.OBJ) contains invalid information or is not specified.

Fatal Errors

Error	Error Message and Description
201	**INVALID COMMAND LINE SYNTAX** **command line** A syntax error is detected in the command line. The command line is displayed up to and including the point of error.
202	**INVALID COMMAND LINE, TOKEN TOO LONG** **command line** The command line contains a token that is too long. The command line is displayed up to and including the point of error.
203	**EXPECTED ITEM MISSING** **command line** An expected item is missing in the command line. The command line is displayed up to and including the point of error.
204	**INVALID KEYWORD** **command line** The invocation line contains an invalid keyword. The command line is displayed up to and including the point of error.
205	**CONSTANT TOO LARGE** **command line** A constant in the invocation line is larger than 0FFFFH. The command line is displayed up to and including the point of error.
206	**INVALID CONSTANT** **command line** A constant in the invocation line is invalid; e.g., a hexadecimal number with a leading letter. The command line is displayed up to and including the point of error.
207	**INVALID NAME** **command line** A module or segment name is invalid. The command line is displayed up to and including the point of error.

Error	Error Message and Description
208	**INVALID FILENAME** **command line** A filename is invalid. The command line is displayed up to and including the point of error.
209	**FILE USED IN CONFLICTING CONTEXTS** **FILE: filename** A specified filename is used for multiple files or used as an input as well as an output file.
210	**I/O ERROR ON INPUT FILE:** **system error message** **FILE: filename** An I/O error is detected by accessing an input file. A detailed error description of the EXCEPTION messages is described afterwards.
211	**I/O ERROR ON OUTPUT FILE:** **system error message** **FILE: filename** An I/O error is detected by accessing an output file. A detailed error description of the EXCEPTION messages is described afterwards.
212	**I/O ERROR ON LISTING FILE:** **system error message** **FILE: filename** An I/O error is detected by accessing a listing file. A detailed error description of the EXCEPTION messages is described afterwards.
213	**I/O ERROR ON WORK FILE:** **system error message** An I/O error is detected by accessing a temporary work file of BL51. A detailed error description of the EXCEPTION messages is described afterwards.
214	**INPUT PHASE ERROR** **MODULE: filename (modulename)** This error occurs when BL51 encounters different data during pass two. This error could be the result of an assembly error.
215	**CHECK SUM ERROR** **MODULE: filename (modulename)** The checksum does not correspond to the contents of the file.
216	**INSUFFICIENT MEMORY** The memory available for the execution of BL51 is used up.

Error	Error Message and Description
217	**NO MODULE TO BE PROCESSED** No module to be processed is found in the invocation line.
218	**NOT AN OBJECT FILE** **FILE: filename** The specified file is not an object file.
219	**NOT AN 8051 OBJECT FILE** **FILE:filename** The specified file is not a valid 8051 object file.
220	**INVALID INPUT MODULE** **FILE: filename** The specified input module is invalid. This error could be the result of an assembler error.
221	**MODULE SPECIFIED MORE THAN ONCE** **command line** The invocation line contains the specified module more than once. The command line is displayed up to and including the point of error.
222	**SEGMENT SPECIFIED MORE THAN ONCE** **command line** The invocation line contains the specified segment more than once. The command line is displayed up to and including the point of error.
224	**DUPLICATE KEYWORD OR CONFLICTING CONTROL** **command line** The same keyword is contained in the invocation line more than once or contradicts with other keywords. The command line is displayed up to and including the point of error.
225	**SEGMENT ADDRESS ARE NOT IN ASCENDING ORDER** **command line** The base addresses for the segments are not displayed in ascending order during the location control. The command line is displayed up to and including the point of error.
226	**SEGMENT ADDRESS INVALID FOR CONTROL** **command line** The base addresses for the segments are invalid for the location control. The command line is displayed up to and including the point of error.

Error	Error Message and Description
227	**PARAMETER OUT OF RANGE** **command line** The specified value for the PAGEWIDTH or PAGELENGTH directive is out of the acceptable range. The command line is displayed up to and including the point of error.
228	**RAMSIZE PARAMETER OUT OF RANGE** **command line** The specified value for the RAMSIZE directive is out of the acceptable range. The command line is displayed up to and including the point of error.
229	**INTERNAL PROCESS ERROR** BL51 detects an internal processing error. Please contact your dealer.
230	**START ADDRESS SPECIFIED MORE THAN ONCE** **command line** The invocation line contains more than one start address for unnamed segment group. The command is displayed up to and including the point of error.
231	**ADDRESS RANGE FOR BANKAREA INCORRECT** **Partial command line** The address space specified with the BANKAREA directive is invalid.
233	**ILLEGAL USE OF * IN OVERLAY CONTROL** **command line** The use of “* ! *” or “* ~ *” with the OVERLAY directive is illegal.

Exceptions

Exception messages are displayed with some error messages. The BL51 code banking linker/locator exception messages that are possible are listed below:

Exception	Exception Message and Description
0021H	**PATH OR FILE NOT FOUND** The specified path or filename is missing.
0026H	**ILLEGAL FILE ACCESS** An attempt was made to write to or delete a write-protected file.
0029H	**ACCESS TO FILE DENIED** The file indicated is a directory.

Exception	Exception Message and Description
002AH	**I/O-ERROR** The drive being written to is either full or the drive was not ready.
0101H	**ILLEGAL CONTEXT** An attempt was made to access a file in an illegal context; e.g., the printer was opened for reading.

Chapter 2. Application Examples

This chapter illustrates some of the linker directives that you may use during project development. These examples use source files created with the C51 compiler and the A51 assembler.

C51 Example

This section describes a short 8051 program, developed with C51 compiler and linked with the BL51 code banking linker/locator. This program demonstrates the concept of modular programming development.

The program calculates the sum of two input numbers and displays the result. Numbers are input with the **getchar** library function and results are output with the **printf** library function. The program consists of three source modules which are translated using the following command lines.

```
C51 CSAMPLE1.C DEBUG

C51 CSAMPLE2.C DEBUG

C51 CSAMPLE3.C DEBUG
```

The **DEBUG** parameter directs the compiler to include complete symbol information in the object file.

After compilation, the files are linked using the BL51 code banking linker/locator. The command line for the linker is:

```
BL51 CSAMPLE1.OBJ, CSAMPLE2.OBJ, CSAMPLE3.OBJ PRECEDE (?DT?CSAMPLE3) IXREF
```

The linker creates an absolute object module that is stored in the file **CSAMPLE1**. This file may be immediately loaded and processed by the dScope-51 simulator or may be used to create an Intel HEX file using the OH51 object to hex converter. In the above linker command line, the **PRECEDE** directive causes the BL51 code banking linker/locator to locate the ?DT?CSAMPLE3 segment before other internal data memory segments. This is explained in detail below. The **IXREF** directive includes a cross reference report of all public and external symbols in the linker listing file.

CSAMPLE1.C Listing File

```
C51 COMPILER,  CSAMPLE1                        10/09/88 14:33:05  PAGE 1


DOS C51 COMPILER, COMPILATION OF MODULE CSAMPLE1
OBJECT MODULE PLACED IN CSAMPLE1.OBJ
COMPILER INVOKED BY: C51 CSAMPLE1.C DEBUG

stmt level    source

  1        /* csample1.c:  C51 Compiler Sample Program */
  2
  3
  4        #include <reg51.h>                       /* define 8051 registers */
  5        #include <stdio.h>                       /* define I/O functions */
  6
  7        extern int getnumber ();
  8        extern output (int);
  9
 10        main ()  {                               /* main program */
 11   1      int number1, number2, result;          /* define operation registers */
 12   1      bit operation;                         /* define operation */
 13   1
 14   1      SCON = 0x52;   /* SCON */              /* setup serial port control */
 15   1      TMOD = 0x20;   /* TMOD */              /* hardware (2400 BAUD @12MHZ) */
 16   1      TCON = 0x69;   /* TCON */
 17   1      TH1 =  0xf3;   /* TH1 */
 18   1
 19   1      printf ("\n\nC-COMPILER-51 demonstration program\n\n");
 20   1
 21   1      while (1)  {                                 /* repeat forever */
 22   2        number1 = getnumber ();                    /* read number1 */
 23   2        number2 = getnumber ();                    /* read number2 */
 24   2        printf ("Input operation: '+' (ADD) or '-' (SUB) ? ");
 25   2        operation = (getchar () == '+');           /* get operation */
 26   2      output (operation ? (number1 + number2)      /* perform operation */
 27   2                          : (number1 - number2) );
 28   2      }
 29   1    }

C51 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

CSAMPLE2.C Listing File

```
C51 COMPILER,  CSAMPLE2                                    10/09/88 14:33:08  PAGE 1


DOS C51 COMPILER, COMPILATION OF MODULE CSAMPLE2
OBJECT MODULE PLACED IN CSAMPLE2.OBJ
COMPILER INVOKED BY: C51 CSAMPLE2.C DEBUG

stmt level    source

   1          /* csample2.c:  C-COMPILER-51 Sample Program */
   2          /* Copyright KEIL ELEKTRONIK GmbH, 1989 */
   3
   4          #include <stdio.h>                      /* define I/O functions */
   5
   6          getline (char *line)  {
   7   1        while ((*line++ = getchar()) != '\n');
   8   1      }
   9
  10          int atoi (char *line)  {
  11   1        bit sign;
  12   1        int number;
  13   1
  14   1        /* skip white space */
  15   1        for ( ; *line == ' ' || *line == '\n' || *line == '\t'; line++);
  16   1
  17   1        /* establish sign */
  18   1        sign = 1;
  19   1        if (*line == '+' || *line == '-')  sign = (*line++ == '+');
  20   1
  21   1        /* compute decimal value */
  22   1        for (number=0; *line >= '0' && *line <= '9'; line++)
  23   1          number = (number * 10) + (*line - '0');
  24   1
  25   1        return (sign ? number : -number);
  26   1      }
  27
  28          unsigned int getnumber ()  {
  29   1        char line [40];
  30   1
  31   1        printf ("Input Number ? ");
  32   1        getline (line);
  33   1        return (atoi (line));
  34   1      }
  35

C51 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

CSAMPLE3.C Listing File

```
C51 COMPILER,  CSAMPLE3                                    10/09/88 14:33:13   PAGE 1


DOS C51 COMPILER, COMPILATION OF MODULE CSAMPLE3
OBJECT MODULE PLACED IN CSAMPLE3.OBJ
COMPILER INVOKED BY: C51 CSAMPLE3.C DEBUG

stmt level    source

  1          /* csample3.c:  C-COMPILER-51 Sample Program */
  2          /* Copyright KEIL ELEKTRONIK GmbH, 1989 */
  3
  4          #include <stdio.h>                    /* define I/O functions */
  5
  6          char dummy_buffer [25];               /* only for demonstration */
  7
  8          output (int number)  {
  9   1        printf ("\nresult: %d\n\n", number);
 10   1      }

C51 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

CSAMPLE Linker/Locator Listing File

```
MCS-51 LINKER / LOCATER  BL51                               DATE  10/09/88    PAGE  1

MS-DOS MCS-51 LINKER / LOCATER  BL51, INVOKED BY:
BL51 CSAMPLE1.OBJ, CSAMPLE2.OBJ, CSAMPLE3.OBJ PRECEDE (?DT?SAMPLE3) IXREF

MEMORY MODEL: SMALL


INPUT MODULES INCLUDED:
  CSAMPLE1.OBJ (CSAMPLE1)
  CSAMPLE2.OBJ (CSAMPLE2)
  CSAMPLE3.OBJ (CSAMPLE3)
  C:\C\C51S.LIB (?C_STARTUP)
  C:\C\C51S.LIB (?C_CLDPTR)
  C:\C\C51S.LIB (?C_CSTPTR)
  C:\C\C51S.LIB (?C_IMUL)
  C:\C\C51S.LIB (?C_PLDIIDATA)
  C:\C\C51S.LIB (PRINTF)
  C:\C\C51S.LIB (GETCHAR)
  C:\C\C51S.LIB (?C_CLDOPTR)
  C:\C\C51S.LIB (?C_CCASE)
  C:\C\C51S.LIB (PUTCHAR)
  C:\C\C51S.LIB (_GETKEY)


LINK MAP OF MODULE:  CSAMPLE1 (CSAMPLE1)

            TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
            -----------------------------------------------------

            * * * * * * *   D A T A   M E M O R Y   * * * * * * *
            REG     0000H     0008H     ABSOLUTE     "REG BANK 0"
            DATA    0008H     0019H     UNIT         ?DT?CSAMPLE3
            DATA    0021H     0001H     BIT_ADDR     ?DB?PRINTF?PRINTF
            BIT     0022H.0   0000H.1   UNIT         ?BI?GETCHAR
            BIT     0022H.1   0000H.2   UNIT         "BIT-GROUP"
                    0022H.3   0000H.5                *** GAP ***
            DATA    0023H     0001H     UNIT         ?DT?GETCHAR
            DATA    0024H     0043H     UNIT         "DATA-GROUP"
            IDATA   0067H     0001H     UNIT         ?STACK

            * * * * * * *   C O D E   M E M O R Y   * * * * * * *
            CODE    0000H     0003H     ABSOLUTE
            CODE    0003H     0052H     UNIT         ?CO?CSAMPLE1
            CODE    0055H     006AH     UNIT         ?PR?MAIN?CSAMPLE1
            CODE    00BFH     0010H     UNIT         ?CO?CSAMPLE2
            CODE    00CFH     00ECH     UNIT         ?PR?ATOI?CSAMPLE2
            CODE    01BBH     002EH     UNIT         ?PR?GETNUMBER?CSAMPLE2
            CODE    01E9H     0016H     UNIT         ?PR?GETLINE?CSAMPLE2
            CODE    01FFH     000EH     UNIT         ?CO?CSAMPLE3
            CODE    020DH     0016H     UNIT         ?PR?OUTPUT?CSAMPLE3
            CODE    0223H     000CH     UNIT         ?C_C51STARTUP
            CODE    022FH     00A8H     UNIT         ?C_LIB_CODE
            CODE    02D7H     0296H     UNIT         ?PR?PRINTF?PRINTF
            CODE    056DH     0013H     UNIT         ?PR?GETCHAR?GETCHAR
            CODE    0580H     0003H     UNIT         ?PR?GETCHAR?UNGETCHAR
            CODE    0583H     0029H     UNIT         ?PR?PUTCHAR?PUTCHAR
            CODE    05ACH     000AH     UNIT         ?PR?_GETKEY?_GETKEY


OVERLAY MAP OF MODULE:   CSAMPLE1 (CSAMPLE1)

SEGMENT                               BIT-GROUP          DATA-GROUP
  +--> CALLING SEGMENT          START    LENGTH     START    LENGTH
-------------------------------------------------------------------
?C_C51STARTUP                   -----    -----      -----    -----
  +--> ?PR?MAIN?CSAMPLE1

?PR?MAIN?SAMPLE1                0022H.1  0000H.1    0024H    0006H
  +--> ?CO?CSAMPLE1
  +--> ?PR?PRINTF?PRINTF
  +--> ?PR?GETNUMBER?CSAMPLE2
  +--> ?PR?GETCHAR?GETCHAR
  +--> ?PR?OUTPUT?CSAMPLE3

?PR?PRINTF?PRINTF               -----    -----      0052H    0014H
  +--> ?C_LIB_CODE
  +--> ?PR?PUTCHAR?PUTCHAR

?PR?PUTCHAR?PUTCHAR             -----    -----      0066H    0001H

?PR?GETNUMBER?CSAMPLE2          -----    -----      002AH    0028H
  +--> ?CO?CSAMPLE2
  +--> ?PR?PRINTF?PRINTF
  +--> ?PR?GETLINE?CSAMPLE2
  +--> ?PR?ATOI?CSAMPLE2

?PR?GETLINE?CSAMPLE2            -----    -----      0052H    0003H
  +--> ?PR?GETCHAR?GETCHAR
  +--> ?C_LIB_CODE

?PR?GETCHAR?GETCHAR             -----    -----      -----    -----
  +--> ?PR?_GETKEY?_GETKEY
  +--> ?PR?PUTCHAR?PUTCHAR

?PR?ATOI?CSAMPLE2               0022H.2  0000H.1    0052H    0005H
  +--> ?C_LIB_CODE

?PR?OUTPUT?CSAMPLE3             -----    -----      002AH    0002H
  +--> ?CO?CSAMPLE3
  +--> ?PR?PRINTF?PRINTF


SYMBOL TABLE OF MODULE:  CSAMPLE1 (CSAMPLE1)

VALUE           TYPE          NAME
----------------------------------
-------         MODULE        CSAMPLE1
C:0055H         PUBLIC        MAIN
-------         PROC          MAIN
D:0024H         SYMBOL        NUMBER1
D:0026H         SYMBOL        NUMBER2
D:0028H         SYMBOL        RESULT
B:0022H.1       SYMBOL        OPERATION
-------         ENDPROC       MAIN
C:0055H         LINE#         10
C:0055H         LINE#         14
C:0058H         LINE#         15
C:005BH         LINE#         16
C:005EH         LINE#         17
C:0061H         LINE#         19
C:0070H         LINE#         21
C:0070H         LINE#         22
C:0077H         LINE#         23
C:007EH         LINE#         24
C:008DH         LINE#         25
C:009BH         LINE#         27
C:00BEH         LINE#         29
-------         ENDMOD        CSAMPLE1

-------         MODULE        CSAMPLE2
C:00CFH         PUBLIC        ATOI
C:01BBH         PUBLIC        GETNUMBER
C:01E9H         PUBLIC        GETLINE
-------         PROC          ATOI
D:0052H         SYMBOL        LINE
B:0022H.2       SYMBOL        SIGN
D:0055H         SYMBOL        NUMBER
-------         ENDPROC       ATOI
C:00CFH         LINE#         10
C:00CFH         LINE#         15
C:0109H         LINE#         18
C:010BH         LINE#         19
C:0142H         LINE#         22
C:0172H         LINE#         23
C:019BH         LINE#         22
C:01A8H         LINE#         25
C:01BAH         LINE#         26
-------         PROC          GETNUMBER
D:002AH         SYMBOL        LINE
-------         ENDPROC       GETNUMBER
C:01BBH         LINE#         28
C:01BBH         LINE#         31
C:01CAH         LINE#         32
C:01D9H         LINE#         33
C:01E8H         LINE#         34
-------         PROC          GETLINE
D:0052H         SYMBOL        LINE
```

```
-------         ENDPROC       GETLINE
C:01E9H         LINE#         6
C:01E9H         LINE#         7
C:01FEH         LINE#         8
-------         ENDMOD        CSAMPLE2

-------         MODULE        CSAMPLE3
D:0008H         PUBLIC        DUMMY_BUFFER
C:020DH         PUBLIC        OUTPUT
-------         PROC          OUTPUT
D:002AH         SYMBOL        NUMBER
-------         ENDPROC       OUTPUT
C:020DH         LINE#         8
C:020DH         LINE#         9
C:0222H         LINE#         10
-------         ENDMOD        CSAMPLE3


INTER-MODULE CROSS-REFERENCE LISTING
------------------------------------

NAME . . . . . . USAGE   MODULE NAMES
-------------------------------------
?ATOI?BIT. . . . BIT;    CSAMPLE2
?ATOI?BYTE . . . DATA;   CSAMPLE2
?C_CCASE . . . . CODE;   ?C_CCASE  PRINTF
?C_CLDOPTR . . . CODE;   ?C_CLDOPTR  PRINTF
?C_CLDPTR. . . . CODE;   ?C_CLDPTR  PRINTF  CSAMPLE2
?C_CSTPTR. . . . CODE;   ?C_CSTPTR  PRINTF  CSAMPLE2
?C_IMUL. . . . . CODE;   ?C_IMUL  CSAMPLE2
?C_PLDIIDATA . . CODE;   ?C_PLDIIDATA  PRINTF  CSAMPLE2
?C_STARTUP . . . CODE;   ?C_STARTUP  CSAMPLE1
?GETLINE?BYTE. . DATA;   CSAMPLE2
?GETNUMBER?BYTE. DATA;   CSAMPLE2
?MAIN?BIT. . . . BIT;    CSAMPLE1
?MAIN?BYTE . . . DATA;   CSAMPLE1
?OUTPUT?BYTE . . DATA;   CSAMPLE3  CSAMPLE1
?PRINTF?BYTE . . DATA;   PRINTF  CSAMPLE1  CSAMPLE2  CSAMPLE3
?PUTCHAR?BYTE. . DATA;   PUTCHAR  GETCHAR  PRINTF
?SPRINTF?BYTE. . DATA;   PRINTF
?UNGETCHAR?BYTE. DATA;   GETCHAR
ATOI . . . . . . CODE;   CSAMPLE2
DUMMY_BUFFER . . DATA;   CSAMPLE3
GETCHAR. . . . . CODE;   GETCHAR  CSAMPLE1  CSAMPLE2
GETLINE. . . . . CODE;   CSAMPLE2
GETNUMBER. . . . CODE;   CSAMPLE2  CSAMPLE1
MAIN . . . . . . CODE;   CSAMPLE1  ?C_STARTUP
OUTPUT . . . . . CODE;   CSAMPLE3  CSAMPLE1
PRINTF . . . . . CODE;   PRINTF  CSAMPLE1  CSAMPLE2  CSAMPLE3
PUTCHAR. . . . . CODE;   PUTCHAR  GETCHAR  PRINTF
SPRINTF. . . . . CODE;   PRINTF
UNGETCHAR. . . . CODE;   GETCHAR
_GETKEY. . . . . CODE;   _GETKEY  GETCHAR
```

In this application, the data segment ?DT?SAMPLE3 is 19H bytes long. Because of its length, this segment can be located in the on-chip data memory only by using the **PRECEDE** directive. Without this directive, the on-chip data memory overflows (because the BIT segment is located first) and the memory space that remains is too small for the STACK (on an 8051/31 CPU).

The following listing shows the data memory usage when the BL51 code banking linker/locator is invoked without the **PRECEDE** directive.

```
            TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
            -----------------------------------------------------
            * * * * * * *   D A T A   M E M O R Y   * * * * * * *
            REG     0000H     0008H     ABSOLUTE     "REG BANK 0"
            DATA    0008H     0001H     UNIT         ?DT?GETCHAR
                    0009H     0017H                  *** GAP ***
            DATA    0020H     0001H     BIT_ADDR     ?DB?PRINTF?PRINTF
            BIT     0021H.0   0000H.1   UNIT         ?BI?GETCHAR
            BIT     0021H.1   0000H.2   UNIT         "BIT-GROUP"
                    0021H.3   0000H.5                *** GAP ***
            DATA    0022H     0019H     UNIT         ?DT?CSAMPLE3
            DATA    003BH     0043H     UNIT         "DATA-GROUP"
            IDATA   007EH     0001H     UNIT         ?STACK
```

Without the **PRECEDE** directive, the ?DT?CSAMPLE3 data segment is located after the BIT segment and the STACK is located at 7Eh.

A51 Example

This section describes a short 8051 program, developed with the A51 assembler and BL51 code banking linker/locator. The program displays the text "PROGRAM TEST" using the **putchar** library function. The program consists of three modules which should be assembled using the following command lines.

```
A51 ASAMPLE1.A51 DEBUG XREF

A51 ASAMPLE2.A51 DEBUG XREF

A51 ASAMPLE3.A51 DEBUG XREF
```

The **XREF** directive causes the A51 assembler to generate a cross reference report of the symbols used in the module. The **DEBUG** directive includes complete symbol information in the object file.

After assembly, the files are linked by the BL51 code banking linker/locator. The command line for the linker is:

```
BL51 ASAMPLE1.OBJ, ASAMPLE2.OBJ, ASAMPLE3.OBJ PRECEDE (VAR1) IXREF
```

The linker creates an absolute object module that is stored in the file **ASAMPLE1**. This file may be immediately loaded and processed by the dScope-51 simulator or may be used to create an Intel HEX file using the OH51 object to hex converter. In the above linker command line, the **PRECEDE** directive causes the BL51 code banking linker/locator to locate the VAR1 segment before other internal data memory segments. The **IXREF** directive includes a cross reference report of all public and external symbols in the linker listing file.

ASAMPLE1.A51 Listing File

```
A51 MACRO ASSEMBLER      ASAMPLE1                       DATE  24/08/87   PAGE     1

MS-DOS MACRO ASSEMBLER A51
OBJECT MODULE PLACED IN ASAMPLE1.OBJ
ASSEMBLER INVOKED BY:  A51 ASAMPLE1.A51 DEBUG XREF


LOC  OBJ            LINE     SOURCE

                       1     NAME    ASAMPLE
                       2
                       3     EXTRN   CODE    (PUT_CRLF, PUTSTRING)
                       4     PUBLIC  TXTBIT
                       5
                       6     PROG    SEGMENT CODE
                       7     CONST   SEGMENT CODE
                       8     VAR1    SEGMENT DATA
                       9     BITVAR  SEGMENT BIT
                      10     STACK   SEGMENT IDATA
                      11
----                  12             RSEG  STACK
0000                  13             DS    10H  ; 16 Bytes Stack
                      14
                      15             CSEG  AT   0
                      16             USING   0  ; Register-Bank 0
                      17     ; Execution starts at address 0 on power-up.
0000 020000   F       18             JMP   START
                      19
----                  20             RSEG  PROG
                      21     ; first set Stack Pointer
0000 758100   F       22     START:  MOV   SP,#STACK-1
                      23
                      24     ; Initialize serial interface
                      25     ; Using TIMER 1 to Generate Baud Rates
                      26     ; Oscillator frequency = 11.059 MHz
0003 758920           27             MOV   TMOD,#00100000B      ;C/T = 0,
                             Mode = 2
0006 758DFD           28             MOV   TH1,#0FDH
0009 D28E             29             SETB  TR1
000B 759852           30             MOV   SCON,#01010010B
                      31
                      32     ; clear TXTBIT to read form CODE-Memory
000E C200     F       33             CLR   TXTBIT
                      34
                      35     ; This is the main program. It is a loop,
                      36     ; which displays the a text on the console.
0010                  37     REPEAT:
                      38     ; type message
0010 900000   F       39             MOV   DPTR,#TXT
0013 120000   F       40             CALL  PUTSTRING
0016 120000   F       41             CALL  PUT_CRLF
                      42     ; repeat
0019 80F5             43             SJMP  REPEAT
                      44     ;
----                  45             RSEG  CONST
0000 54455354         46     TXT:    DB    'TEST PROGRAM',00H
0004 2050524F
0008 4752414D
000C 00
                      47
                      48     ; only for demonstration
----                  49             RSEG  VAR1
0000                  50     DUMMY:  DS    21H
                      51
                      52     ; TXTBIT = 0 read text from CODE  Memory
                      53     ; TXTBIT = 1 read text from XDATA Memory
----                  54             RSEG  BITVAR
0000                  55     TXTBIT: DBIT  1
                      56
                      57             END
                      58



XREF SYMBOL TABLE LISTING
---- ------ ----- -------

N A M E     T Y P E  V A L U E   ATTRIBUTES / REFERENCES

BITVAR . .  B SEG    0000H            REL=UNIT  9# 54
CONST. . .  C SEG    000DH            REL=UNIT  7# 45
DUMMY. . .  D ADDR   0000H   R        SEG=VAR1   50#
PROG . . .  C SEG    001BH            REL=UNIT  6# 20
PUTSTRING.  C ADDR   ----        EXT     3 40
PUT_CRLF .  C ADDR   ----        EXT     3 41
REPEAT . .  C ADDR   0010H   R        SEG=PROG   37# 43
ASAMPLE . .   ----   ----                1
SCON . . .  D ADDR   0098H   A           30
SP . . . .  D ADDR   0081H   A           22
STACK. . .  I SEG    0010H            REL=UNIT  10# 12 22
START. . .  C ADDR   0000H   R        SEG=PROG   18 22#
TH1. . . .  D ADDR   008DH   A           28
TMOD . . .  D ADDR   0089H   A           27
TR1. . . .  B ADDR   0088H.6 A           29
TXT. . . .  C ADDR   0000H   R        SEG=CONST   39 46#
TXTBIT . .  B ADDR   0000H.0 R   PUB  SEG=BITVAR   4 33 55#
VAR1 . . .  D SEG    0021H            REL=UNIT  8# 49


REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

ASAMPLE2.A51 Listing File

```
A51 MACRO ASSEMBLER      ASAMPLE2                       DATE  24/08/87   PAGE      1

MS-DOS MACRO ASSEMBLER A51
OBJECT MODULE PLACED IN ASAMPLE2.OBJ
ASSEMBLER INVOKED BY:  A51 ASAMPLE2.A51 DEBUG XREF


LOC  OBJ            LINE     SOURCE

                       1     NAME    STRING_IO
                       2     ;
                       3     EXTRN   BIT    (TXTBIT)
                       4     EXTRN   CODE   (PUTCHAR)
                       5     PUBLIC  PUT_CRLF, PUTSTRING
                       6
                       7     STRING_ROUTINES  SEGMENT  CODE
                       8
----                   9             RSEG  STRING_ROUTINES
                      10     ; This routine outputs a CR and a LF
  000D                11     CR equ  0DH                ; carriage return
  000A                12     LF equ  0AH                ; line feed
                      13
0000                  14     PUT_CRLF:
0000 740D             15             MOV   A,#CR
0002 120000   F       16             CALL  PUTCHAR
0005 740A             17             MOV   A,#LF
0007 120000   F       18             CALL  PUTCHAR
000A 22               19             RET
                      20
                      21     ; Routine outputs a null-terminated string whose
                      22     ; address is given in DPTR. The string can be
                      23     ; located in CODE or XDATA memory depending on
```

```
                        24     ; the value of TXTBIT.
                        25
000B                    26     PUTSTRING:
                        27     ; check TXTBIT
000B 200004   F         28             JB    TXTBIT,PS1
000E E4                 29             CLR   A
000F 93                 30             MOVC  A,@A+DPTR
0010 8001               31             SJMP  PS2
0012 E0                 32     PS1:    MOVX  A,@DPTR
0013 6006               33     PS2:    JZ    EXIT
0015 120000   F         34             CALL  PUTCHAR
0018 A3                 35             INC   DPTR
0019 80F0               36             SJMP  PUTSTRING
001B 22                 37     EXIT:   RET
                        38
                        39             END
                        40


XREF SYMBOL TABLE LISTING
---- ------ ----- -------

N A M E          T Y P E  V A L U E   ATTRIBUTES / REFERENCES

CR. . . . . . .  N NUMB   000DH   A           11# 15
EXIT. . . . . .  C ADDR   001BH   R        SEG=STRING_ROUTINES   33 37#
LF. . . . . . .  N NUMB   000AH   A           12# 17
PS1 . . . . . .  C ADDR   0012H   R        SEG=STRING_ROUTINES   28 32#
PS2 . . . . . .  C ADDR   0013H   R        SEG=STRING_ROUTINES   31 33#
PUTCHAR . . . .  C ADDR   ----        EXT     4 16 18 34
PUTSTRING . . .  C ADDR   000BH   R   PUB  SEG=STRING_ROUTINES   5 26# 36
PUT_CRLF. . . .  C ADDR   0000H   R   PUB  SEG=STRING_ROUTINES   5 14#
STRING_IO . . .    ----   ----                1
STRING_ROUTINES  C SEG    0000H            REL=UNIT  7# 9
TXTBIT. . . . .  B ADDR   ----        EXT     3 28


REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

ASAMPLE3.A51 Listing File

```
A51 MACRO ASSEMBLER      ASAMPLE3                       DATE  24/08/87   PAGE      1

MS-DOS MACRO ASSEMBLER A51
OBJECT MODULE PLACED IN ASAMPLE3.OBJ
ASSEMBLER INVOKED BY:  A51 ASAMPLE3.A51 DEBUG XREF

LOC  OBJ            LINE     SOURCE

                         1     NAME    CHAR_IO
                         2     ;
                         3     PUBLIC  PUTCHAR
                         4
                         5     CHAR_ROUTINES  SEGMENT  CODE
                         6     VAR2           SEGMENT  DATA
                         7
----                     8             RSEG  CHAR_ROUTINES
                         9
                        10     ; This routine outputs a single character to
                        11     ; console. The character is given in A.
0000                    12     PUTCHAR:
0000 3099FD             13             JNB  TI,$
0003 C299               14             CLR  TI
0005 F599               15             MOV  SBUF,A
0007 22                 16             RET
                        17
```

```
                      18
                      19     ; only for demonstration
----                  20             RSEG  VAR2
0000                  21     DUMMY:  DS    40H
                      22
                      23
                      24             END
                      25


XREF SYMBOL TABLE LISTING
---- ------ ----- -------

N A M E        T Y P E  V A L U E   ATTRIBUTES / REFERENCES

CHAR_IO . . .    ----   ----                1
CHAR_ROUTINES  C SEG    0008H            REL=UNIT  5# 8
DUMMY . . . .  D ADDR   0000H   R        SEG=VAR2   21#
PUTCHAR . . .  C ADDR   0000H   R   PUB  SEG=CHAR_ROUTINES   3 12#
SBUF. . . . .  D ADDR   0099H   A           15
TI. . . . . .  B ADDR   0098H.1 A           13 14
VAR2. . . . .  D SEG    0000H            REL=UNIT  6# 20


REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

ASAMPLE Linker/Locator Listing File

```
MCS-51 LINKER / LOCATER  BL51                          DATE  24/08/87   PAGE     1


MS-DOS MCS-51 LINKER / LOCATER  BL51, INVOKED BY:
BL51 ASAMPLE1.OBJ, ASAMPLE2.OBJ, ASAMPLE3.OBJ PRECEDE (VAR1) IXREF

INPUT MODULES INCLUDED:
  ASAMPLE1.OBJ (ASAMPLE)
  ASAMPLE2.OBJ (STRING_IO)
  ASAMPLE3.OBJ (CHAR_IO)


LINK MAP OF MODULE:  ASAMPLE1 (ASAMPLE)

            TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
            -----------------------------------------------------

            * * * * * * *   D A T A   M E M O R Y   * * * * * * *
            REG     0000H     0008H     ABSOLUTE     "REG BANK 0"
            DATA    0008H     0021H     UNIT         VAR1
            BIT     0029H.0   0000H.1   UNIT         BITVAR
                    0029H.1   0000H.7                *** GAP ***
            DATA    002AH     0040H     UNIT         VAR2
            IDATA   006AH     0010H     UNIT         STACK

            * * * * * * *   C O D E   M E M O R Y   * * * * * * *
            CODE    0000H     0003H     ABSOLUTE
            CODE    0003H     001BH     UNIT         PROG
            CODE    001EH     000DH     UNIT         CONST
            CODE    002BH     001CH     UNIT         STRING_ROUTINES
            CODE    0047H     0008H     UNIT         CHAR_ROUTINES


SYMBOL TABLE OF MODULE:  ASAMPLE1 (ASAMPLE)

VALUE           TYPE          NAME
----------------------------------
```

```
-------         MODULE        ASAMPLE
B:0029H.0       SEGMENT       BITVAR
C:001EH         SEGMENT       CONST
D:0008H         SYMBOL        DUMMY
C:0003H         SEGMENT       PROG
C:0013H         SYMBOL        REPEAT
D:0098H         SYMBOL        SCON
D:0081H         SYMBOL        SP
I:006AH         SEGMENT       STACK
C:0003H         SYMBOL        START
D:008DH         SYMBOL        TH1
D:0089H         SYMBOL        TMOD
B:0088H.6       SYMBOL        TR1
C:001EH         SYMBOL        TXT
B:0029H.0       PUBLIC        TXTBIT
D:0008H         SEGMENT       VAR1
-------         ENDMOD        ASAMPLE
-------         MODULE        STRING_IO
N:000DH         SYMBOL        CR
C:0046H         SYMBOL        EXIT
N:000AH         SYMBOL        LF
C:003DH         SYMBOL        PS1
C:003EH         SYMBOL        PS2
C:0036H         PUBLIC        PUTSTRING
C:002BH         PUBLIC        PUT_CRLF
C:002BH         SEGMENT       STRING_ROUTINES
-------         ENDMOD        STRING_IO

-------         MODULE        CHAR_IO
C:0047H         SEGMENT       CHAR_ROUTINES
D:002AH         SYMBOL        DUMMY
C:0047H         PUBLIC        PUTCHAR
D:0099H         SYMBOL        SBUF
B:0098H.1       SYMBOL        TI
D:002AH         SEGMENT       VAR2
-------         ENDMOD        CHAR_IO


INTER-MODULE CROSS-REFERENCE LISTING
------------------------------------

NAME . . . . USAGE   MODULE NAMES
---------------------------------

PUTCHAR. . . CODE;   CHAR_IO  STRING_IO
PUTSTRING. . CODE;   STRING_IO  ASAMPLE
PUT_CRLF . . CODE;   STRING_IO  ASAMPLE
TXTBIT . . . BIT;    ASAMPLE  STRING_IO
```

Code Banking Examples

This section includes application examples that use code banking with the BL51 code banking linker/locator.

Example 1. Code Banking with C51

The following C51 example shows how to compile and link a program using multiple code banks.

The program begins with function **main** in **C_ROOT.C**. The **main** function calls functions in other code banks. These functions, in turn, call functions in yet different code banks. The **printf** function outputs the number of the code bank in each function.

The program can be translated using the following commands:

```
C51 C_ROOT.C   DEBUG OBJECTEXTEND

C51 C_BANK0.C  DEBUG OBJECTEXTEND

C51 C_BANK1.C  DEBUG OBJECTEXTEND

C51 C_BANK2.C  DEBUG OBJECTEXTEND
```

All program modules are translated using the C51 compiler. **C_ROOT.C** contains the **main** function and is located in the common area. **C_BANK0.C**, **C_BANK1.C**, and **C_BANK2.C** contain the bank functions and are located in the bank area. The BL51 code banking linker/locator is invoked as follows:

```
BL51 COMMON{C_ROOT.OBJ}, BANK0{C_BANK0.OBJ}, &
>>  BANK1{C_BANK1.OBJ}, BANK2{C_BANK2.OBJ} &
>>  BANKAREA(8000H,0FFFFH)
```

The **BANKAREA (8000H, 0FFFFH)** directive defines the address space 80000H to 0FFFFH as the area for code banks. The **COMMON** directive places the **C_ROOT.OBJ** module in the common area. The **BANK0**, **BANK1**, and **BANK2** directives place modules in bank 0, 1, and 2 respectively.

The BL51 code banking linker/locator creates a listing file, **C_ROOT.M51**, which contains information about memory allocation and about the intra-bank jump table that is generated. BL51 also creates the output module, **C_ROOT**, that is stored in banked OMF format. You must use the OC51 banked object file converter to convert the banked OMF file into standard OMF files. OMF files can be loaded with the dScope simulator or an in-circuit emulator. Invoke the OC51 banked object file converter as follows:

```
OC51 C_ROOT
```

For this example program, the OC51 banked object file converter produces three standard OMF-51 files from **C_ROOT**. They are listed in the following table.

Filename	Contents
C_ROOT.B00	All information (including symbols) for code bank 0 and the common area.
C_ROOT.B01	Information for code bank 1 and the common area.
C_ROOT.B02	Information for code bank 2 and the common area.

You can create Intel HEX files for each of these OMF-51 files by using the OH51 object to hex converter. The Intel HEX files you create with OH51 contain complete information for each code bank including the common area. Intel HEX files can be generated using the following OH51 object to hex converter command line.

```
OH51 C_ROOT.B00 HEXFILE (C_ROOT.H00)

OH51 C_ROOT.B01 HEXFILE (C_ROOT.H01)

OH51 C_ROOT.B02 HEXFILE (C_ROOT.H02)
```

Following are listings of the C source files and the linker map file.

C_ROOT.C Listing File

```
C51 COMPILER,  C_ROOT                                   11/03/91  17:33:34  PAGE 1


DOS C51 COMPILER, COMPILATION OF MODULE C_ROOT
OBJECT MODULE PLACED IN C_ROOT.OBJ
COMPILER INVOKED BY: G:\C51.EXE C_ROOT.C DEBUG OBJECTEXTEND

stmt level    source

   1          #include <stdio.h>
   2          #include <reg51.h>
   3
   4          extern void func0(void);
   5          extern void func1(void);
   6
   7          void main(void) {
   8   1
   9   1        /* Initialize serial interface to 2400 baud @12MHz */
  10   1        SCON = 0x52;    /* SCON */
  11   1        TMOD = 0x20;    /* TMOD */
  12   1        TCON = 0x69;    /* TCON */
  13   1        TH1 =  0xf3;    /* TH1 */
  14   1
  15   1        printf("Main program calls a function in bank 0 \n.");
  16   1        func0();
  17   1        printf("Main program calls a function in bank 1 \n.");
  18   1        func1();
  19   1
  20   1        while(1);
  21   1      }

MODULE INFORMATION:   STATIC OVERLAYABLE
   CODE SIZE        =     39    ----
   CONSTANT SIZE    =     84    ----
   XDATA SIZE       =   ----    ----
   PDATA SIZE       =   ----    ----
   DATA SIZE        =   ----    ----
   IDATA SIZE       =   ----    ----
   BIT SIZE         =   ----    ----
END OF MODULE INFORMATION.

C51 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

C_BANK0.C Listing File

```
C51 COMPILER,  C_BANK0                                    11/03/91  17:33:35  PAGE 1


DOS C51 COMPILER, COMPILATION OF MODULE C_BANK0
OBJECT MODULE PLACED IN C_BANK0.OBJ
COMPILER INVOKED BY: G:\C51.EXE C_BANK0.C DEBUG OBJECTEXTEND

stmt level    source

   1          #include <stdio.h>
   2
   3          extern void func2(void);
   4
   5          void func0(void) {
   6   1        printf("Function in bank 0 calls a function in bank 2 \n.");
   7   1        func2();
   8   1      }

MODULE INFORMATION:   STATIC OVERLAYABLE
   CODE SIZE        =     13    ----
   CONSTANT SIZE    =     48    ----
   XDATA SIZE       =   ----    ----
   PDATA SIZE       =   ----    ----
   DATA SIZE        =   ----    ----
   IDATA SIZE       =   ----    ----
   BIT SIZE         =   ----    ----
END OF MODULE INFORMATION.

C51 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

C_BANK1.C Listing File

```
C51 COMPILER,  C_BANK1                                    11/03/91  17:33:36  PAGE 1


DOS C51 COMPILER, COMPILATION OF MODULE C_BANK1
OBJECT MODULE PLACED IN C_BANK1.OBJ
COMPILER INVOKED BY: G:\C51.EXE C_BANK1.C DEBUG OBJECTEXTEND

stmt level    source

   1          #include <stdio.h>
   2
   3          extern void func2(void);
   4
   5          void func1(void) {
   6   1        printf("Function in bank  1 calls a function in bank 2 \n.");
   7   1        func2();
   8   1      }

MODULE INFORMATION:   STATIC OVERLAYABLE
   CODE SIZE        =     13    ----
   CONSTANT SIZE    =     48    ----
   XDATA SIZE       =   ----    ----
   PDATA SIZE       =   ----    ----
   DATA SIZE        =   ----    ----
   IDATA SIZE       =   ----    ----
   BIT SIZE         =   ----    ----
END OF MODULE INFORMATION.

C51 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

C_BANK2.C Listing File

```
C51 COMPILER,  C_BANK2                                  11/03/91  17:33:36  PAGE 1


DOS C51 COMPILER, COMPILATION OF MODULE C_BANK2
OBJECT MODULE PLACED IN C_BANK2.OBJ
COMPILER INVOKED BY: G:\C51.EXE C_BANK2.C DEBUG OBJECTEXTEND

stmt level    source

   1          #include <stdio.h>
   2
   3          void func2(void) {
   4   1        printf("This is a function in bank 2! \n.");
   5   1      }

MODULE INFORMATION:   STATIC OVERLAYABLE
   CODE SIZE        =     10    ----
   CONSTANT SIZE    =     32    ----
   XDATA SIZE       =   ----    ----
   PDATA SIZE       =   ----    ----
   DATA SIZE        =   ----    ----
   IDATA SIZE       =   ----    ----
   BIT SIZE         =   ----    ----
END OF MODULE INFORMATION.

C51 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

C_ROOT Linker/Locator Listing File

```
BL51 BANKED LINKER/LOCATER                              11/03/91  17:33:37  PAGE 1

MS-DOS BL51 BANKED LINKER/LOCATER, INVOKED BY:
F:\C51P\BIN\BL51.EXE COMMON {C_ROOT.OBJ}, BANK0 {C_BANK0.OBJ}, BANK1 {C_BANK1.OBJ},
>> BANK2 {C_BANK2.OBJ} BANKAREA (8000H,0FFFFH)

MEMORY MODEL: SMALL


INPUT MODULES INCLUDED:
  C_ROOT.OBJ (C_ROOT)
  C_BANK0.OBJ (C_BANK0)
  C_BANK1.OBJ (C_BANK1)
  C_BANK2.OBJ (C_BANK2)
  F:\C51P\LIB\L51_BANK.OBJ (?BANK?SWITCHING)
  F:\C51P\LIB\C51S.LIB (?C_STARTUP)
  F:\C51P\LIB\C51S.LIB (PRINTF)
  F:\C51P\LIB\C51S.LIB (?C_CLDPTR)
  F:\C51P\LIB\C51S.LIB (?C_CLDOPTR)
  F:\C51P\LIB\C51S.LIB (?C_CSTPTR)
  F:\C51P\LIB\C51S.LIB (?C_PLDIIDATA)
  F:\C51P\LIB\C51S.LIB (?C_CCASE)
  F:\C51P\LIB\C51S.LIB (PUTCHAR)



LINK MAP OF MODULE:  C_ROOT (C_ROOT)

            TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
            -----------------------------------------------------

            * * * * * * *   D A T A   M E M O R Y   * * * * * * *
            REG     0000H     0008H     ABSOLUTE     "REG BANK 0"
            DATA    0008H     0014H     UNIT         "DATA_GROUP"
                    001CH     0004H                  *** GAP ***
            BIT     0020H.0   0001H.1   UNIT         "BIT_GROUP"
                    0021H.1   0000H.7                *** GAP ***
```

2

```
            IDATA   0022H     0001H     UNIT         ?STACK

            * * * * * * *   C O D E   M E M O R Y   * * * * * * *
            CODE    0000H     0003H     ABSOLUTE
            CODE    0003H     0027H     UNIT         ?PR?MAIN?C_ROOT
            CODE    002AH     0054H     UNIT         ?CO?C_ROOT
            CODE    007EH     0030H     UNIT         ?CO?C_BANK0
            CODE    00AEH     0030H     UNIT         ?CO?C_BANK1
            CODE    00DEH     0020H     UNIT         ?CO?C_BANK2
            CODE    00FEH     0187H     INBLOCK      ?BANK?SELECT
            CODE    0285H     000CH     UNIT         ?C_C51STARTUP
            CODE    0291H     0027H     UNIT         ?PR?PUTCHAR?PUTCHAR
                    02B8H     0048H                  *** GAP ***
            CODE    0300H     007FH     PAGE         ?BANK?SWITCH
            CODE    037FH     032BH     UNIT         ?PR?PRINTF?PRINTF
            CODE    06AAH     0094H     UNIT         ?C_LIB_CODE

            * * * * * * *   C O D E   B A N K   0   * * * * * * *
                    0000H     8000H                  *** GAP ***
            BANK0   8000H     000DH     UNIT         ?PR?FUNC0?C_BANK0

            * * * * * * *   C O D E   B A N K   1   * * * * * * *
                    0000H     8000H                  *** GAP ***
            BANK1   8000H     000DH     UNIT         ?PR?FUNC1?C_BANK1

            * * * * * * *   C O D E   B A N K   2   * * * * * * *
                    0000H     8000H                  *** GAP ***
            BANK2   8000H     000AH     UNIT         ?PR?FUNC2?C_BANK2


OVERLAY MAP OF MODULE:   C_ROOT (C_ROOT)


SEGMENT                           BIT-GROUP          DATA-GROUP
  +--> CALLED SEGMENT          START    LENGTH     START    LENGTH
------------------------------------------------------------------
?C_C51STARTUP                  -----    -----      -----    -----
  +--> ?PR?MAIN?C_ROOT

?PR?MAIN?C_ROOT                -----    -----      -----    -----
  +--> ?CO?C_ROOT
  +--> ?PR?PRINTF?PRINTF


  +--> ?PR?FUNC0?C_BANK0
  +--> ?PR?FUNC1?C_BANK1

?PR?PRINTF?PRINTF              0020H.0  0001H.1    0008H    0014H
  +--> ?C_LIB_CODE
  +--> ?PR?PUTCHAR?PUTCHAR

?PR?FUNC0?C_BANK0              -----    -----      -----    -----
  +--> ?CO?C_BANK0
  +--> ?PR?PRINTF?PRINTF
  +--> ?PR?FUNC2?C_BANK2

?PR?FUNC2?C_BANK2              -----    -----      -----    -----
  +--> ?CO?C_BANK2
  +--> ?PR?PRINTF?PRINTF

?PR?FUNC1?C_BANK1              -----    -----      -----    -----
  +--> ?CO?C_BANK1
  +--> ?PR?PRINTF?PRINTF
  +--> ?PR?FUNC2?C_BANK2

INTRABANK CALL TABLE OF MODULE:  C_ROOT (C_ROOT)

ADDRESS    FUNCTION NAME
------------------------
 0275H     FUNC0
 027AH     FUNC1
 027FH     FUNC2
```

```
SYMBOL TABLE OF MODULE:  C_ROOT (C_ROOT)

  VALUE           TYPE          NAME
  ----------------------------------

  -------         MODULE        C_ROOT
  C:0000H         SYMBOL        _ICE_DUMMY_
  C:0003H         PUBLIC        main
  D:0098H         PUBLIC        SCON
  D:0089H         PUBLIC        TMOD
  D:0088H         PUBLIC        TCON
  D:008DH         PUBLIC        TH1
  -------         PROC          MAIN
  C:0003H         LINE#         7
  C:0003H         LINE#         10
  C:0006H         LINE#         11
  C:0009H         LINE#         12
  C:000CH         LINE#         13
  C:000FH         LINE#         15
  C:0018H         LINE#         16
  C:001BH         LINE#         17
  C:0024H         LINE#         18
  C:0027H         LINE#         20
  C:0029H         LINE#         21
  -------         ENDPROC       MAIN
  -------         ENDMOD        C_ROOT

  -------         MODULE        C_BANK0
  C:0000H         SYMBOL        _ICE_DUMMY_
 C0:8000H         PUBLIC        func0

  -------         PROC BANK=0   FUNC0
 C0:8000H         LINE#         5
 C0:8000H         LINE#         6
 C0:8009H         LINE#         7
 C0:800CH         LINE#         8
  -------         ENDPROC       FUNC0
  -------         ENDMOD        C_BANK0

  -------         MODULE        C_BANK1
  C:0000H         SYMBOL        _ICE_DUMMY_
 C1:8000H         PUBLIC        func1

  -------         PROC BANK=1   FUNC1
 C1:8000H         LINE#         5

 C1:8000H         LINE#         6
 C1:8009H         LINE#         7
 C1:800CH         LINE#         8
  -------         ENDPROC       FUNC1
  -------         ENDMOD        C_BANK1

  -------         MODULE        C_BANK2
  C:0000H         SYMBOL        _ICE_DUMMY_
 C2:8000H         PUBLIC        func2

  -------         PROC BANK=2   FUNC2
 C2:8000H         LINE#         3
 C2:8000H         LINE#         4
 C2:8009H         LINE#         5
  -------         ENDPROC       FUNC2
  -------         ENDMOD        C_BANK2

  -------         MODULE        ?BANK?SWITCHING
  N:0010H         PUBLIC        ?B_NBANKS
  N:0000H         PUBLIC        ?B_MODE
  D:0090H         PUBLIC        ?B_CURRENTBANK
  N:0078H         PUBLIC        ?B_MASK
  C:026EH         PUBLIC        _SWITCHBANK
  C:00FEH         PUBLIC        ?B_BANK0
  C:0115H         PUBLIC        ?B_BANK1
  C:012CH         PUBLIC        ?B_BANK2
  C:0143H         PUBLIC        ?B_BANK3
  C:015AH         PUBLIC        ?B_BANK4
  C:0171H         PUBLIC        ?B_BANK5
  C:0188H         PUBLIC        ?B_BANK6
  C:019FH         PUBLIC        ?B_BANK7
  C:01B6H         PUBLIC        ?B_BANK8
  C:01CDH         PUBLIC        ?B_BANK9
  C:01E4H         PUBLIC        ?B_BANK10
  C:01FBH         PUBLIC        ?B_BANK11
  C:0212H         PUBLIC        ?B_BANK12
  C:0229H         PUBLIC        ?B_BANK13
  C:0240H         PUBLIC        ?B_BANK14
  C:0257H         PUBLIC        ?B_BANK15
  -------         ENDMOD        ?BANK?SWITCHING

  -------         MODULE        PRINTF
  D:0008H         PUBLIC        ?_PRINTF517?BYTE
  D:0008H         PUBLIC        ?_SPRINTF517?BYTE
  D:0008H         PUBLIC        ?_PRINTF?BYTE
  D:0008H         PUBLIC        ?_SPRINTF?BYTE
  C:03E4H         PUBLIC        _PRINTF
  C:03DEH         PUBLIC        _SPRINTF
  C:03E4H         PUBLIC        _PRINTF517
  C:03DEH         PUBLIC        _SPRINTF517
  -------         ENDMOD        PRINTF

  -------         MODULE        ?C_CLDPTR
  C:06AAH         PUBLIC        ?C_CLDPTR
  -------         ENDMOD        ?C_CLDPTR

  -------         MODULE        ?C_CLDOPTR
  C:06C5H         PUBLIC        ?C_CLDOPTR
  -------         ENDMOD        ?C_CLDOPTR

  -------         MODULE        ?C_CSTPTR
  C:06F4H         PUBLIC        ?C_CSTPTR
  -------         ENDMOD        ?C_CSTPTR

  -------         MODULE        ?C_PLDIIDATA
  C:0708H         PUBLIC        ?C_PLDIIDATA
  -------         ENDMOD        ?C_PLDIIDATA

  -------         MODULE        ?C_CCASE
  C:0718H         PUBLIC        ?C_CCASE
  -------         ENDMOD        ?C_CCASE

  -------         MODULE        PUTCHAR
  C:0291H         PUBLIC        _PUTCHAR
  -------         ENDMOD        PUTCHAR

LINK/LOCATE RUN COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

Example 2. Code Banking with Constants

This example shows how to place constants in code banks. You can use this technique to place messages or large tables in code banks other than the one in which your program resides.

You use the BL51 code banking linker/locator to locate constant segments in particular code banks. Segment names for constant data have the general format ?CO?*modulename* where *modulename* is the name of the source file the constant data is declared.

In your C51 programs, when you access constant data that is in a different segment, you must manually ensure that the proper code bank is used when accessing that constant data. You so this with the **switchbank** function. This function is defined in the **L51_BANK.A51** source module.

This example uses three source files: **C_PROG.C**, **C_MESS0.C**, and **C_MESS1.C**. These source files are compiled and linked using the following commands.

```
C51 C_PROG.C    DEBUG OBJECTEXTEND

C51 C_MESS0.C   DEBUG OBJECTEXTEND

C51 C_MESS1.C   DEBUG OBJECTEXTEND

BL51 C_PROG.OBJ, C_MESS0.OBJ, C_MESS1.OBJ &
>>  BANKAREA(8000H,0FFFFH) &
>>  BANK0(?CO?C_MESS0 (8000H)) BANK1(?CO?C_MESS1 (8000H))

OC51 C_PROG

OH51 C_PROG.B00 HEXFILE (C_PROG.H00)

OH51 C_PROG.B01 HEXFILE (C_PROG.H01)
```

The OMF-51 files, **C_PROG.B00** and **C_PROG.B01**, can be loaded with the dScope simulator or an in-circuit emulator.

The Intel HEX files, **C_PROG.H00** and **C_PROG.H01**, can be used with an EPROM programmer.

Following are listings of the C51 source files and the linker map file.

C_PROG.C Listing File

```
C51 COMPILER,  C_PROG                                      12/03/91  10:22:36  PAGE 1


DOS C51 COMPILER, COMPILATION OF MODULE C_PROG
OBJECT MODULE PLACED IN C_PROG.OBJ
COMPILER INVOKED BY: G:\C51.EXE C_PROG.C DEBUG OBJECTEXTEND

stmt level    source

   1          #include <stdio.h>
   2          #include <reg51.h>
   3
   4          extern char *message0[];
   5          extern char *message1[];
   6          extern switchbank (unsigned char);
   7
   8          void main(void) {
   9   1
  10   1      /* Initialise serial interface to 2400 baud @12MHz */
  11   1        SCON = 0x52;    /* SCON */
  12   1        TMOD = 0x20;    /* TMOD */
  13   1        TCON = 0x69;    /* TCON */
  14   1        TH1 =  0xf3;    /* TH1 */
  15   1
  16   1        switchbank(0);                /* Switch to code bank 0 */
  17   1        printf(message0[0]);
  18   1        switchbank(1);                /* Switch to code bank 1 */
  19   1        printf(message1[0]);
  20   1
  21   1        while(1);
  22   1      }
```

C_MESS0.C Listing File

```
C51 COMPILER,  C_MESS0                                    12/03/91  10:28:22  PAGE 1


DOS C51 COMPILER, COMPILATION OF MODULE C_MESS0
OBJECT MODULE PLACED IN C_MESS0.OBJ
COMPILER INVOKED BY: G:\C51.EXE C_MESS0.C DEBUG OBJECTEXTEND

stmt level    source

   1          code char *message0[] = {
   2            "This is a message from code bank 0\n.",
   3            "This is another text."
   4          };
```

C_MESS1.C Listing File

```
C51 COMPILER,  C_MESS1                                    12/03/91  10:28:22  PAGE 1


DOS C51 COMPILER, COMPILATION OF MODULE C_MESS1
OBJECT MODULE PLACED IN C_MESS1.OBJ
COMPILER INVOKED BY: G:\C51.EXE C_MESS1.C DEBUG OBJECTEXTEND

stmt level    source

   1          code char *message1[] = {
   2            "This is a message from code bank 1\n.",
   3            "This is another text."
   4          };
```

C_PROG Linker/Locator Listing File

```
BL51 BANKED LINKER/LOCATER                                13/03/91  09:10:54  PAGE 1


MS-DOS BL51 BANKED LINKER/LOCATER, INVOKED BY:
F:\C51P\BIN\BL51.EXE C_PROG.OBJ, C_MESS0.OBJ, C_MESS1.OBJ BANKAREA (8000H, 0FFFFH)
BANK0 (?CO?C_MESS0 (8000H)) BANK1 (?C
>> O?C_MESS1 (8000H))

MEMORY MODEL: SMALL


INPUT MODULES INCLUDED:
  C_PROG.OBJ (C_PROG)
  C_MESS0.OBJ (C_MESS0)
  C_MESS1.OBJ (C_MESS1)
  F:\C51P\LIB\L51_BANK.OBJ (?BANK?SWITCHING)
  F:\C51P\LIB\C51S.LIB (?C_STARTUP)
  F:\C51P\LIB\C51S.LIB (PRINTF)
  F:\C51P\LIB\C51S.LIB (?C_CLDPTR)
  F:\C51P\LIB\C51S.LIB (?C_CLDOPTR)
  F:\C51P\LIB\C51S.LIB (?C_CSTPTR)
  F:\C51P\LIB\C51S.LIB (?C_PLDIIDATA)
  F:\C51P\LIB\C51S.LIB (?C_CCASE)
  F:\C51P\LIB\C51S.LIB (PUTCHAR)


LINK MAP OF MODULE:  C_PROG (C_PROG)

            TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
            -----------------------------------------------------

            * * * * * * *   D A T A   M E M O R Y   * * * * * * *
            REG     0000H     0008H     ABSOLUTE     "REG BANK 0"
            DATA    0008H     0014H     UNIT         "DATA_GROUP"
                    001CH     0004H                  *** GAP ***
            BIT     0020H.0   0001H.1   UNIT         "BIT_GROUP"
                    0021H.1   0000H.7                *** GAP ***
            IDATA   0022H     0001H     UNIT         ?STACK

            * * * * * * *   C O D E   M E M O R Y   * * * * * * *
            CODE    0000H     0003H     ABSOLUTE
            CODE    0003H     003BH     UNIT         ?PR?MAIN?C_PROG
            CODE    003EH     0178H     INBLOCK      ?BANK?SELECT
            CODE    01B6H     000CH     UNIT         ?C_C51STARTUP
            CODE    01C2H     0027H     UNIT         ?PR?PUTCHAR?PUTCHAR
                    01E9H     0017H                  *** GAP ***
            CODE    0200H     007FH     PAGE         ?BANK?SWITCH
            CODE    027FH     032BH     UNIT         ?PR?PRINTF?PRINTF
            CODE    05AAH     0094H     UNIT         ?C_LIB_CODE

            * * * * * * *   C O D E   B A N K   0   * * * * * * *
                    0000H     8000H                  *** GAP ***
            BANK0   8000H     003FH     UNIT         ?CO?C_MESS0

            * * * * * * *   C O D E   B A N K   1   * * * * * * *
                    0000H     8000H                  *** GAP ***
            BANK1   8000H     003FH     UNIT         ?CO?C_MESS1


OVERLAY MAP OF MODULE:   C_PROG (C_PROG)

SEGMENT                           BIT-GROUP          DATA-GROUP
  +--> CALLED SEGMENT          START    LENGTH     START    LENGTH
------------------------------------------------------------------
?C_C51STARTUP                  -----    -----      -----    -----
  +--> ?PR?MAIN?C_PROG

?PR?MAIN?C_PROG                -----    -----      -----    -----
  +--> ?CO?C_MESS0
  +--> ?PR?PRINTF?PRINTF
  +--> ?CO?C_MESS1

?PR?PRINTF?PRINTF              0020H.0  0001H.1    0008H    0014H
  +--> ?C_LIB_CODE
  +--> ?PR?PUTCHAR?PUTCHAR


SYMBOL TABLE OF MODULE:  C_PROG (C_PROG)

  VALUE           TYPE          NAME
  ----------------------------------
  -------         MODULE        C_PROG
  C:0000H         SYMBOL        _ICE_DUMMY_
  C:0003H         PUBLIC        main
  D:0098H         PUBLIC        SCON
  D:0089H         PUBLIC        TMOD
  D:0088H         PUBLIC        TCON
  D:008DH         PUBLIC        TH1
  -------         PROC          MAIN
  C:0003H         LINE#         8
  C:0003H         LINE#         11
```

```
 C:0006H         LINE#         12
 C:0009H         LINE#         13
 C:000CH         LINE#         14
 C:000FH         LINE#         16
 C:0014H         LINE#         17
 C:0025H         LINE#         18
 C:002AH         LINE#         19
 C:003BH         LINE#         21
 C:003DH         LINE#         22
 -------         ENDPROC       MAIN
 -------         ENDMOD        C_PROG

 -------         MODULE        C_MESS0
 C:0000H         SYMBOL        _ICE_DUMMY_
C0:8039H         PUBLIC        message0
 -------         ENDMOD        C_MESS0

 -------         MODULE        C_MESS1
 C:0000H         SYMBOL        _ICE_DUMMY_
C1:8039H         PUBLIC        message1
 -------         ENDMOD        C_MESS1

 -------         MODULE        ?BANK?SWITCHING
 N:0010H         PUBLIC        ?B_NBANKS
 N:0000H         PUBLIC        ?B_MODE
 D:0090H         PUBLIC        ?B_CURRENTBANK
 N:0078H         PUBLIC        ?B_MASK
 C:01AEH         PUBLIC        _SWITCHBANK
 C:003EH         PUBLIC        ?B_BANK0
 C:0055H         PUBLIC        ?B_BANK1
 C:006CH         PUBLIC        ?B_BANK2
 C:0083H         PUBLIC        ?B_BANK3
 C:009AH         PUBLIC        ?B_BANK4
 C:00B1H         PUBLIC        ?B_BANK5
 C:00C8H         PUBLIC        ?B_BANK6
 C:00DFH         PUBLIC        ?B_BANK7
 C:00F6H         PUBLIC        ?B_BANK8
 C:010DH         PUBLIC        ?B_BANK9
 C:0124H         PUBLIC        ?B_BANK10
 C:013BH         PUBLIC        ?B_BANK11
 C:0152H         PUBLIC        ?B_BANK12
 C:0169H         PUBLIC        ?B_BANK13
 C:0180H         PUBLIC        ?B_BANK14
 C:0197H         PUBLIC        ?B_BANK15
 -------         ENDMOD        ?BANK?SWITCHING

 -------         MODULE        PRINTF
 D:0008H         PUBLIC        ?_PRINTF517?BYTE
 D:0008H         PUBLIC        ?_SPRINTF517?BYTE
 D:0008H         PUBLIC        ?_PRINTF?BYTE
 D:0008H         PUBLIC        ?_SPRINTF?BYTE
 C:02E4H         PUBLIC        _PRINTF
 C:02DEH         PUBLIC        _SPRINTF
 C:02E4H         PUBLIC        _PRINTF517
 C:02DEH         PUBLIC        _SPRINTF517
 -------         ENDMOD        PRINTF

 -------         MODULE        ?C_CLDPTR
 C:05AAH         PUBLIC        ?C_CLDPTR
 -------         ENDMOD        ?C_CLDPTR


 -------         MODULE        ?C_CLDOPTR
 C:05C5H         PUBLIC        ?C_CLDOPTR
 -------         ENDMOD        ?C_CLDOPTR

 -------         MODULE        ?C_CSTPTR
 C:05F4H         PUBLIC        ?C_CSTPTR
 -------         ENDMOD        ?C_CSTPTR

 -------         MODULE        ?C_PLDIIDATA
 C:0608H         PUBLIC        ?C_PLDIIDATA
 -------         ENDMOD        ?C_PLDIIDATA
 -------         MODULE        ?C_CCASE
 C:0618H         PUBLIC        ?C_CCASE
 -------         ENDMOD        ?C_CCASE

 -------         MODULE        PUTCHAR
 C:01C2H         PUBLIC        _PUTCHAR
 -------         ENDMOD        PUTCHAR

LINK/LOCATE RUN COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

Example 3. Placing Specific Functions in Code Banks

This example shows how you can locate a single function in a specific code bank. To do this, you use directives on the command line for the BL51 code banking linker/locator.

This example locates an interrupt function, **timer0**, in the common area. The segment name for this function is ?PR?TIMER0?C_MODUL. This example also locates an initialization function, **tinit**, in code bank 1. The segment name for this function is ?PR?TINIT?C_MODUL.

Both functions are contained in **C_MODUL.C**. The following commands were used to compile and link this example.

```
C51 C_MODUL.C   DEBUG OBJECTEXTEND

BL51 BANK0{C_MODUL.OBJ} BANKAREA(8000H,0FFFFH) &
>>  COMMON (?PR?TIMER0?C_MODUL) &
>>  BANK1(?PR?TINIT?C_MODUL (8000H))

   OC51 C_MODUL

   OH51 C_MODUL.B00 HEXFILE (C_MODUL.H00)

   OH51 C_MODUL.B01 HEXFILE (C_MODUL.H01)
```

The OMF-51 files, **C_MODUL.B00** and **C_MODUL.B01**, can be loaded with the dScope simulator or an in-circuit emulator.

The Intel HEX files, **C_MODUL.H00** and **C_MODUL.H01**, can be used with an EPROM programmer.

Following are listings of the C51 source file, **C_MODUL.C**, and the linker map file.

C_MODUL.C Listing File

```
C51 COMPILER,  C_MODUL                                     11/03/91  17:33:52  PAGE 1


DOS C51 COMPILER, COMPILATION OF MODULE C_MODUL
OBJECT MODULE PLACED IN C_MODUL.OBJ
COMPILER INVOKED BY: G:\C51.EXE C_MODUL.C DEBUG OBJECTEXTEND

stmt level    source

   1          #include <stdio.h>
   2          #include <reg51.h>
   3
   4          unsigned long msec;                    /* Millisecond counter    */
   5          unsigned char intcycle;                /* Interrupt cycle counter */
   6
   7          /**********************************************/
   8          /* Timer 0 interrupt service function         */
   9          /* executes each 250us @ 12 MHz crystal clock */
  10          /**********************************************/
  11          timer0() interrupt 1 using 1      /* int vector at 000BH, reg. bank 1*/
  12          {
  13   1        if (++intcycle == 4)  {            /* 1 msec = 4* 250 usec cycle  */
  14   2          intcycle = 0;
  15   2          msec++;
  16   2        }
  17   1      }
  18
  19
  20          /***************************/
  21          /* setup timer 0 interrupt */
  22          /***************************/
  23          tinit ()  {
  24   1        TH0 = -250;                        /* Set timer period        */
  25   1        TL0 = -250;
  26   1        TMOD = TMOD | 0x02;                /* Select mode 2           */
  27   1        TR0 = 1;                           /* Start timer 0           */
  28   1        ET0 = 1;                           /* Enable timer 0 interrupt*/
  29   1        EA  = 1;                           /* Global interrupt enable */
  30   1      }
  31
  32          void main(void) {
  33   1      /* INITIALIZE SERIAL INTERFACE TO 2400 BAUD @12MHz */
  34   1        SCON = 0x52;    /* SCON */
  35   1        TMOD = 0x20;    /* TMOD */
  36   1        TCON = 0x69;    /* TCON */
  37   1        TH1 =  0xf3;    /* TH1 */
  38   1
  39   1        tinit ();                           /* Initialize timer 0 */
  40   1        while(1)  {
  41   2          printf ("MSEC=%lu\r", msec);
  42   2        }
  43   1      }
```

C_MODUL Linker/Locator Listing File

```
BL51 BANKED LINKER/LOCATER                                13/03/91  09:11:19  PAGE 1


MS-DOS BL51 BANKED LINKER/LOCATER, INVOKED BY:
F:\C51P\BIN\BL51.EXE BANK0 {C_MODUL.OBJ} COMMON (?PR?TIMER0?C_MODUL) BANK1
(?PR?TINIT?C_MODUL
>> (8000H)) BANKAREA (8000H, 0FFFFH)

MEMORY MODEL: SMALL

INPUT MODULES INCLUDED:
  C_MODUL.OBJ (C_MODUL)
  F:\C51P\LIB\L51_BANK.OBJ (?BANK?SWITCHING)
  F:\C51P\LIB\C51S.LIB (?C_STARTUP)
  F:\C51P\LIB\C51S.LIB (?C_LADD)
  F:\C51P\LIB\C51S.LIB (?C_ISTACK)
  F:\C51P\LIB\C51S.LIB (PRINTF)
  F:\C51P\LIB\C51S.LIB (?C_CLDPTR)
  F:\C51P\LIB\C51S.LIB (?C_CLDOPTR)
  F:\C51P\LIB\C51S.LIB (?C_CSTPTR)
  F:\C51P\LIB\C51S.LIB (?C_LACC)
  F:\C51P\LIB\C51S.LIB (?C_PLDIIDATA)
  F:\C51P\LIB\C51S.LIB (?C_CCASE)
  F:\C51P\LIB\C51S.LIB (PUTCHAR)


LINK MAP OF MODULE:  C_MODUL (C_MODUL)


            TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
            -----------------------------------------------------

            * * * * * * *   D A T A   M E M O R Y   * * * * * * *
            REG     0000H     0008H     ABSOLUTE     "REG BANK 0"
            REG     0008H     0008H     ABSOLUTE     "REG BANK 1"
            DATA    0010H     0005H     UNIT         ?DT?C_MODUL
            DATA    0015H     0005H     UNIT         ?C_LIB_DATA
                    001AH     0006H                  *** GAP ***
            BIT     0020H.0   0001H.1   UNIT         "BIT_GROUP"
                    0021H.1   0000H.7                *** GAP ***
            DATA    0022H     0014H     UNIT         "DATA_GROUP"
            IDATA   0036H     0001H     UNIT         ?STACK

            * * * * * * *   C O D E   M E M O R Y   * * * * * * *
            CODE    0000H     0003H     ABSOLUTE
                    0003H     0008H                  *** GAP ***
            CODE    000BH     0003H     ABSOLUTE
            CODE    000EH     0040H     UNIT         ?PR?TIMER0?C_MODUL
            CODE    004EH     000AH     UNIT         ?CO?C_MODUL
            CODE    0058H     0182H     INBLOCK      ?BANK?SELECT
            CODE    01DAH     000CH     UNIT         ?C_C51STARTUP
                    01E6H     001AH                  *** GAP ***
            CODE    0200H     007FH     PAGE         ?BANK?SWITCH
            CODE    027FH     00E6H     UNIT         ?C_LIB_CODE
            CODE    0365H     032BH     UNIT         ?PR?PRINTF?PRINTF
            CODE    0690H     0027H     UNIT         ?PR?PUTCHAR?PUTCHAR

            * * * * * * *   C O D E   B A N K   0   * * * * * * *
                    0000H     8000H                  *** GAP ***
            BANK0   8000H     0027H     UNIT         ?PR?MAIN?C_MODUL

            * * * * * * *   C O D E   B A N K   1   * * * * * * *
                    0000H     8000H                  *** GAP ***
            BANK1   8000H     0010H     UNIT         ?PR?TINIT?C_MODUL


OVERLAY MAP OF MODULE:   C_MODUL (C_MODUL)


SEGMENT                           BIT-GROUP          DATA-GROUP
  +--> CALLED SEGMENT          START    LENGTH     START    LENGTH
------------------------------------------------------------------
?PR?TIMER0?C_MODUL             -----    -----      -----    -----
  +--> ?C_LIB_CODE

?C_51STARTUP                   -----    -----      -----    -----
  +--> ?PR?MAIN?C_MODUL

?PR?MAIN?C_MODUL               -----    -----      -----    -----
  +--> ?PR?TINIT?C_MODUL
  +--> ?CO?C_MODUL
  +--> ?PR?PRINTF?PRINTF

?PR?PRINTF?PRINTF              0020H.0  0001H.1    0022H    0014H
```

```
  +--> ?C_LIB_CODE
  +--> ?PR?PUTCHAR?PUTCHAR


INTRABANK CALL TABLE OF MODULE:  C_MODUL (C_MODUL)

ADDRESS    FUNCTION NAME
------------------------
 01CFH     TINIT
 01D4H     ?C_START (= MAIN)


SYMBOL TABLE OF MODULE:  C_MODUL (C_MODUL)

  VALUE           TYPE          NAME
  ----------------------------------

  -------         MODULE        C_MODUL
  C:0000H         SYMBOL        _ICE_DUMMY_
  B:00A8H.7       PUBLIC        EA
 C0:8000H         PUBLIC        main
  D:0010H         PUBLIC        msec
 C1:8000H         PUBLIC        tinit
  D:0098H         PUBLIC        SCON
  D:0089H         PUBLIC        TMOD
  D:0088H         PUBLIC        TCON
  B:00A8H.1       PUBLIC        ET0
  D:008CH         PUBLIC        TH0
  D:008DH         PUBLIC        TH1
  D:008AH         PUBLIC        TL0
  C:000EH         PUBLIC        timer0
  B:0088H.4       PUBLIC        TR0
  D:0014H         PUBLIC        intcycle
  -------         PROC          TIMER0
  C:000EH         LINE#         11
  C:001BH         LINE#         13
  C:0022H         LINE#         14
  C:0025H         LINE#         15
  C:0043H         LINE#         16
  C:0043H         LINE#         17
  -------         ENDPROC       TIMER0

  -------         PROC BANK=1   TINIT
 C1:8000H         LINE#         23
 C1:8000H         LINE#         24
 C1:8003H         LINE#         25
 C1:8006H         LINE#         26
 C1:8009H         LINE#         27
 C1:800BH         LINE#         28
 C1:800DH         LINE#         29
 C1:800FH         LINE#         30
  -------         ENDPROC       TINIT

  -------         PROC BANK=0   MAIN
 C0:8000H         LINE#         32
 C0:8000H         LINE#         34
 C0:8003H         LINE#         35
 C0:8006H         LINE#         36
 C0:8009H         LINE#         37
 C0:800CH         LINE#         39
 C0:800FH         LINE#         40
 C0:800FH         LINE#         41
 C0:8024H         LINE#         42
 C0:8026H         LINE#         43
  -------         ENDPROC       MAIN
  -------         ENDMOD        C_MODUL


  -------         MODULE        ?BANK?SWITCHING
  N:0010H         PUBLIC        ?B_NBANKS
  N:0000H         PUBLIC        ?B_MODE
  D:0090H         PUBLIC        ?B_CURRENTBANK
  N:0078H         PUBLIC        ?B_MASK
  C:01C8H         PUBLIC        _SWITCHBANK
  C:0058H         PUBLIC        ?B_BANK0
  C:006FH         PUBLIC        ?B_BANK1
  C:0086H         PUBLIC        ?B_BANK2
  C:009DH         PUBLIC        ?B_BANK3
  C:00B4H         PUBLIC        ?B_BANK4
  C:00CBH         PUBLIC        ?B_BANK5
  C:00E2H         PUBLIC        ?B_BANK6
  C:00F9H         PUBLIC        ?B_BANK7
  C:0110H         PUBLIC        ?B_BANK8
  C:0127H         PUBLIC        ?B_BANK9
  C:013EH         PUBLIC        ?B_BANK10
  C:0155H         PUBLIC        ?B_BANK11
  C:016CH         PUBLIC        ?B_BANK12
  C:0183H         PUBLIC        ?B_BANK13
  C:019AH         PUBLIC        ?B_BANK14
  C:01B1H         PUBLIC        ?B_BANK15
  -------         ENDMOD        ?BANK?SWITCHING

  -------         MODULE        ?C_LADD
  C:027FH         PUBLIC        ?C_LADD
  -------         ENDMOD        ?C_LADD

  -------         MODULE        ?C_ISTACK
  D:0015H         PUBLIC        ?C_DSTKLEVEL
  C:0292H         PUBLIC        ?C_LPUSH
  C:02B1H         PUBLIC        ?C_LPULL
  C:02B9H         PUBLIC        ?C_LSTKDEC
  -------         ENDMOD        ?C_ISTACK

  -------         MODULE        PRINTF
  D:0022H         PUBLIC        ?_PRINTF517?BYTE
  D:0022H         PUBLIC        ?_SPRINTF517?BYTE
  D:0022H         PUBLIC        ?_PRINTF?BYTE
  D:0022H         PUBLIC        ?_SPRINTF?BYTE
  C:03CAH         PUBLIC        _PRINTF
  C:03C4H         PUBLIC        _SPRINTF
  C:03CAH         PUBLIC        _PRINTF517
  C:03C4H         PUBLIC        _SPRINTF517
  -------         ENDMOD        PRINTF

  -------         MODULE        ?C_CLDPTR
  C:02D1H         PUBLIC        ?C_CLDPTR
  -------         ENDMOD        ?C_CLDPTR

  -------         MODULE        ?C_CLDOPTR
  C:02ECH         PUBLIC        ?C_CLDOPTR
  -------         ENDMOD        ?C_CLDOPTR

  -------         MODULE        ?C_CSTPTR
  C:031BH         PUBLIC        ?C_CSTPTR
  -------         ENDMOD        ?C_CSTPTR

  -------         MODULE        ?C_PLDIIDATA
  C:032FH         PUBLIC        ?C_PLDIIDATA
  -------         ENDMOD        ?C_PLDIIDATA

  -------         MODULE        ?C_CCASE
  C:033FH         PUBLIC        ?C_CCASE
  -------         ENDMOD        ?C_CCASE

  -------         MODULE        PUTCHAR
  C:0690H         PUBLIC        _PUTCHAR
  -------         ENDMOD        PUTCHAR

LINK/LOCATE RUN COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

Example 4. Code Banking with PL/M-51

The following PL/M-51 examples shows how to compile and link a PL/M-51 program using multiple code banks. The function of this example is similar to that shown in "Example 1. Code Banking with C51" on page 112.

The program begins with the procedure in **P_ROOT.P51**. This routine calls routines in other code banks which, in turn, call routines in yet different code banks.

The PL/M-51 programs are compiled using the following commands.

```
PLM51 P_ROOT.P51  DEBUG

PLM51 P_BANK0.P51 DEBUG

PLM51 P_BANK1.P51 DEBUG

PLM51 P_BANK2.P51 DEBUG
```

In this example, **P_ROOT.OBJ** is located in the common area and **P_BANK0.OBJ**, **P_BANK1.OBJ**, and **P_BANK2.OBJ** are located in the bank area.

NOTE
*The PL/M-51 runtime library, **PLM51.LIB**, must be included in the linkage. You must either specify a path to the directory in which this library is stored, or you must include it directly in the linker command line.*

The BL51 code banking linker/locator is invoked as follows:

```
BL51 COMMON{P_ROOT.OBJ}, BANK0{P_BANK0.OBJ}, &
>>  BANK1{P_BANK1.OBJ}, BANK2{P_BANK2.OBJ} &
>>  BANKAREA(8000H,0FFFFH)
```

The **BANKAREA (8000H, 0FFFFH)** directive defines the address space 80000H to 0FFFFH as the area for code banks. The **COMMON** directive places the **P_ROOT.OBJ** module in the common area. The **BANK0**, **BANK1**, and **BANK2** directives place modules in bank 0, 1, and 2 respectively.

The BL51 code banking linker/locator creates a listing file, **P_ROOT.M51**, which contains information about memory allocation and about the intra-bank jump table that is generated. BL51 also creates the output module, **P_ROOT**, that is stored in banked OMF format. You must use the OC51 banked object file converter to convert the banked OMF file into standard OMF files. OMF files can be loaded with the dScope simulator or an in-circuit emulator. Invoke the OC51 banked object file converter as follows:

```
OC51 P_ROOT
```

For this example program, the OC51 banked object file converter produces three standard OMF-51 files from **P_ROOT**. They are listed in the following table.

Filename	Contents
P_ROOT.B00	All information (including symbols) for code bank 0 and the common area.
P_ROOT.B01	Information for code bank 1 and the common area.
P_ROOT.B02	Information for code bank 2 and the common area.

You can create Intel HEX files for each of these OMF-51 files by using the OH51 object to hex converter. The Intel HEX files you create with OH51 contain complete information for each code bank including the common area. Intel HEX files can be generated using the following OH51 object to hex converter command line.

```
OH51 P_ROOT.B00 HEXFILE (P_ROOT.H00)

OH51 P_ROOT.B01 HEXFILE (P_ROOT.H01)

OH51 P_ROOT.B02 HEXFILE (P_ROOT.H02)
```

Following are listings of the PL/M-51 source files and the linker map file.

P_ROOT.P51 Listing File

```
PL/M-51 COMPILER                                        03/11/91           PAGE   1


DOS 4.0 (038-N) PL/M-51
COMPILER INVOKED BY:  F:\C51P\BIN\PLM51.EXE P_ROOT.P51 DEBUG

   1   1      P_ROOT: DO;

   2   2      FUNC0: PROCEDURE EXTERNAL; END;
   4   2      FUNC1: PROCEDURE EXTERNAL; END;


                /* Start of main program */

                /* Main program calls a function in bank 0  */
   6   1        CALL FUNC0;
```

```
                /* Main program calls a function in bank 1  */
   7   1        CALL FUNC1;

   8   2        DO WHILE (1); END;

  10   1      END;


MODULE INFORMATION:                (STATIC+OVERLAYABLE)
    CODE SIZE                      = 0008H         8D
    CONSTANT SIZE                  = 0000H         0D
    DIRECT VARIABLE SIZE           =   00H+00H     0D+  0D
    INDIRECT VARIABLE SIZE         =   00H+00H     0D+  0D
    BIT SIZE                       =   00H+00H     0D+  0D
    BIT-ADDRESSABLE SIZE           =   00H+00H     0D+  0D
    AUXILIARY VARIABLE SIZE        = 0000H         0D
    MAXIMUM STACK SIZE             = 0004H         4D
    REGISTER-BANK(S) USED:           0
    17 LINES READ
    0 PROGRAM ERROR(S)
END OF PL/M-51 COMPILATION
```

P_BANK0.P51 Listing File

```
PL/M-51 COMPILER                                        03/11/91           PAGE   1


DOS 4.0 (038-N) PL/M-51
COMPILER INVOKED BY:  F:\C51P\BIN\PLM51.EXE P_BANK0.P51 DEBUG

   1   1      P_BANK0: DO;

   2   2      FUNC2: PROCEDURE EXTERNAL; END;

   4   2      FUNC0: PROCEDURE PUBLIC;
                /* Function in bank 0 calls a function in bank 2 */
   5   2        CALL FUNC2;
   6   2      END;

   7   1      END;



MODULE INFORMATION:                (STATIC+OVERLAYABLE)
    CODE SIZE                      = 0004H         4D
    CONSTANT SIZE                  = 0000H         0D
    DIRECT VARIABLE SIZE           =   00H+00H     0D+  0D
    INDIRECT VARIABLE SIZE         =   00H+00H     0D+  0D
    BIT SIZE                       =   00H+00H     0D+  0D
    BIT-ADDRESSABLE SIZE           =   00H+00H     0D+  0D
    AUXILIARY VARIABLE SIZE        = 0000H         0D
    MAXIMUM STACK SIZE             = 0002H         2D
    REGISTER-BANK(S) USED:           0
    11 LINES READ
    0 PROGRAM ERROR(S)
END OF PL/M-51 COMPILATION
```

P_BANK1.P51 Listing File

```
PL/M-51 COMPILER                                        03/11/91           PAGE   1


DOS 4.0 (038-N) PL/M-51
COMPILER INVOKED BY:  F:\C51P\BIN\PLM51.EXE P_BANK1.P51 DEBUG
```

```
   1   1      P_BANK1: DO;

   2   2      FUNC2: PROCEDURE EXTERNAL; END;

   4   2      FUNC1: PROCEDURE PUBLIC;
                /* Function in bank 1 calls a function in bank 2 */
   5   2        CALL FUNC2;
   6   2      END;

   7   1      END;



MODULE INFORMATION:                (STATIC+OVERLAYABLE)
    CODE SIZE                      = 0004H         4D
    CONSTANT SIZE                  = 0000H         0D
    DIRECT VARIABLE SIZE           =   00H+00H     0D+  0D
    INDIRECT VARIABLE SIZE         =   00H+00H     0D+  0D
    BIT SIZE                       =   00H+00H     0D+  0D
    BIT-ADDRESSABLE SIZE           =   00H+00H     0D+  0D
    AUXILIARY VARIABLE SIZE        = 0000H         0D
    MAXIMUM STACK SIZE             = 0002H         2D
    REGISTER-BANK(S) USED:           0
    11 LINES READ
    0 PROGRAM ERROR(S)
END OF PL/M-51 COMPILATION
```

P_BANK2.P51 Listing File

```
PL/M-51 COMPILER                                       03/11/91            PAGE   1


DOS 4.0 (038-N) PL/M-51
COMPILER INVOKED BY:  F:\C51P\BIN\PLM51.EXE P_BANK2.P51 DEBUG

   1   1      P_BANK2: DO;

   2   2      FUNC2: PROCEDURE PUBLIC;
                /* This is a function in bank 2. */
   3   2      END;

   4   1      END;

MODULE INFORMATION:                (STATIC+OVERLAYABLE)
    CODE SIZE                      = 0001H         1D
    CONSTANT SIZE                  = 0000H         0D
    DIRECT VARIABLE SIZE           =   00H+00H     0D+  0D
    INDIRECT VARIABLE SIZE         =   00H+00H     0D+  0D
    BIT SIZE                       =   00H+00H     0D+  0D
    BIT-ADDRESSABLE SIZE           =   00H+00H     0D+  0D
    AUXILIARY VARIABLE SIZE        = 0000H         0D
    MAXIMUM STACK SIZE             = 0002H         2D
    REGISTER-BANK(S) USED:           0
    7 LINES READ
    0 PROGRAM ERROR(S)
END OF PL/M-51 COMPILATION
```

P_ROOT Linker/Locator Listing File

```
BL51 BANKED LINKER/LOCATER                               11/03/91  17:34:03  PAGE 1


MS-DOS BL51 BANKED LINKER/LOCATER, INVOKED BY:
F:\C51P\BIN\BL51.EXE COMMON {P_ROOT.OBJ}, BANK0 {P_BANK0.OBJ}, BANK1 {P_BANK1.OBJ},
BANK2
>> {P_BANK2.OBJ} BANKAREA (8000H,0FFFFH)
```

```
MEMORY MODEL: SMALL (PL/M-51)

INPUT MODULES INCLUDED:
  P_ROOT.OBJ (P_ROOT)
  P_BANK0.OBJ (P_BANK0)
  P_BANK1.OBJ (P_BANK1)
  P_BANK2.OBJ (P_BANK2)
  F:\C51P\LIB\L51_BANK.OBJ (?BANK?SWITCHING)
  F:\C51P\LIB\PLM51.LIB (?PIV0R)


LINK MAP OF MODULE:  P_ROOT (P_ROOT)


            TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
            -----------------------------------------------------

            * * * * * * *   D A T A   M E M O R Y   * * * * * * *
            REG     0000H     0008H     ABSOLUTE     "REG BANK 0"
            IDATA   0008H     0001H     UNIT         ?STACK

            * * * * * * *   C O D E   M E M O R Y   * * * * * * *
            CODE    0000H     0003H     ABSOLUTE
            CODE    0003H     0008H     INBLOCK      ?P_ROOT?PR
            CODE    000BH     0187H     INBLOCK      ?BANK?SELECT
            CODE    0192H     0009H     UNIT         ?PIV0RS
                    019BH     0065H                  *** GAP ***
            CODE    0200H     007FH     PAGE         ?BANK?SWITCH

            * * * * * * *   C O D E   B A N K   0   * * * * * * *
                    0000H     8000H                  *** GAP ***
            BANK0   8000H     0004H     INBLOCK      ?P_BANK0?PR

            * * * * * * *   C O D E   B A N K   1   * * * * * * *
                    0000H     8000H                  *** GAP ***
            BANK1   8000H     0004H     INBLOCK      ?P_BANK1?PR

            * * * * * * *   C O D E   B A N K   2   * * * * * * *
                    0000H     8000H                  *** GAP ***
            BANK2   8000H     0001H     INBLOCK      ?P_BANK2?PR


OVERLAY MAP OF MODULE:   P_ROOT (P_ROOT)


SEGMENT
  +--> CALLED SEGMENT
---------------------
?PIV0RS
  +--> ?P_ROOT?PR

?P_ROOT?PR
  +--> ?P_BANK0?PR
  +--> ?P_BANK1?PR

?P_BANK0?PR
  +--> ?P_BANK2?PR

?P_BANK1?PR
  +--> ?P_BANK2?PR


INTRABANK CALL TABLE OF MODULE:  P_ROOT (P_ROOT)

ADDRESS    FUNCTION NAME
------------------------
 0182H     FUNC0
 0187H     FUNC1
 018CH     FUNC2


SYMBOL TABLE OF MODULE:  P_ROOT (P_ROOT)
```

```
 VALUE           TYPE          NAME
 ----------------------------------

 -------         MODULE        P_ROOT
 C:0003H         SYMBOL        P_ROOT
 C:0003H         LINE#         6
 C:0006H         LINE#         7
 C:0009H         LINE#         8
 C:0009H         LINE#         9
 C:000BH         LINE#         10
 -------         ENDMOD        P_ROOT

 -------         MODULE        P_BANK0
 C:8000H         PUBLIC        FUNC0
 C:8004H         SYMBOL        P_BANK0

 -------         PROC BANK=0   FUNC0
 -------         ENDPROC       FUNC0
C0:8000H         LINE#         4
C0:8000H         LINE#         5
C0:8003H         LINE#         6
C0:8004H         LINE#         7
 -------         ENDMOD        P_BANK0

 -------         MODULE        P_BANK1
 C:8000H         PUBLIC        FUNC1
 C:8004H         SYMBOL        P_BANK1

 -------         PROC BANK=1   FUNC1
 -------         ENDPROC       FUNC1
C1:8000H         LINE#         4
C1:8000H         LINE#         5
C1:8003H         LINE#         6
C1:8004H         LINE#         7
 -------         ENDMOD        P_BANK1

 -------         MODULE        P_BANK2
 C:8000H         PUBLIC        FUNC2
 C:8001H         SYMBOL        P_BANK2

 -------         PROC BANK=2   FUNC2
 -------         ENDPROC       FUNC2
C2:8001H         LINE#         1
C2:8000H         LINE#         2
C2:8000H         LINE#         3
C2:8001H         LINE#         4
 -------         ENDMOD        P_BANK2

 -------         MODULE        ?BANK?SWITCHING
 N:0010H         PUBLIC        ?B_NBANKS
 N:0000H         PUBLIC        ?B_MODE
 D:0090H         PUBLIC        ?B_CURRENTBANK
 N:0078H         PUBLIC        ?B_MASK
 C:017BH         PUBLIC        _SWITCHBANK
 C:000BH         PUBLIC        ?B_BANK0
 C:0022H         PUBLIC        ?B_BANK1
 C:0039H         PUBLIC        ?B_BANK2
 C:0050H         PUBLIC        ?B_BANK3
 C:0067H         PUBLIC        ?B_BANK4
 C:007EH         PUBLIC        ?B_BANK5
 C:0095H         PUBLIC        ?B_BANK6
 C:00ACH         PUBLIC        ?B_BANK7
 C:00C3H         PUBLIC        ?B_BANK8
 C:00DAH         PUBLIC        ?B_BANK9
 C:00F1H         PUBLIC        ?B_BANK10
 C:0108H         PUBLIC        ?B_BANK11
 C:011FH         PUBLIC        ?B_BANK12
 C:0136H         PUBLIC        ?B_BANK13
 C:014DH         PUBLIC        ?B_BANK14
 C:0164H         PUBLIC        ?B_BANK15
 -------         ENDMOD        ?BANK?SWITCHING
```

```
LINK/LOCATE RUN COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

Chapter 3. LIB51 Library Manager

The LIB51 library manager allows you to create and maintain library files. A library file is a formatted collection of one or more object files. Library files provide a convenient method of referencing a large number of object files and can be used by the L51 linker/locator.

The LIB51 library manager allows you to create library files, add object modules, remove object modules, and list library file contents. The LIB51 library manager can be controlled interactively or from the command line.

Using LIB51

To invoke the LIB51 library manager from the DOS prompt, type **LIB51** along with an optional library manager command. The command line must be entered according to the following format:

```
LIB51 ⟦command⟧
```

where ***command*** may be a single library manager command. To enter more than one command, append the ampersand character (**&**) to the end of the LIB51 library manager command line.

Interactive Mode

If no *command* is entered on the command line, or if the ampersand character is included at the end of the line, the LIB51 library manager enters interactive mode. The LIB51 library manager displays an asterisk character (*****) to signal that it is in interactive mode and is waiting for input.

Any of the LIB51 library manager commands may be entered on the command line or after the ***** prompt when in interactive mode.

Type **EXIT** to leave the LIB51 library manager interactive mode.

Command Summary

The following table lists the commands that are available for the LIB51 library manager. All of these commands are described in detail in the sections that follow.

Command	Abbreviation	Description
ADD	**A**	adds an object module to the library file.
CREATE	**C**	creates a new library file.
DELETE	**D**	removes an object module from a library file.
EXIT	**E**	exits the interactive mode of the LIB51 library manager.
HELP	**H**	displays help information for the LIB51 library manager.
LIST	**L**	displays module and public symbol information stored in a library file.

Creating a Library

The **CREATE** command directs the LIB51 library manager to create a new, empty library file. The **CREATE** command may be entered on the command line, or at the ***** prompt in interactive mode, and must have the following format:

```
CREATE libfile
```

where *libfile* is the name of the library file to create. The name of the library file must include the file extension. Usually, **.LIB** is the extension that is used for library files.

Example:

```
LIB51 CREATE MYFILE.LIB

* CREATE FASTMATH.LIB
```

Adding Object Modules

The **ADD** command instructs the LIB51 library manager to add one or more object modules to a specified library file. The **ADD** command must be entered in the following format:

```
ADD filename ⟦(modulename, …)⟧ ⟦, …⟧ TO libfile
```

where

filename is the name of an object file or library file. You may specify several files for each **ADD** command. Each file must be separated by a comma.

modulename is the name of a module in a library file. If you do not want to add the entire contents of a library, you may select the modules that you want to add. Module names are specified immediately following the ***filename***, must be enclosed in parentheses, and must be separated by commas.

libfile is the name of an existing library file. The specified object modules are added to this library.

The **ADD** command may be entered on the command line or after the ***** prompt in interactive mode as shown in the following example.

```
LIB51 ADD MOD1.OBJ, UTIL.LIB(FPMUL, FPDIV) TO NEW.LIB

* ADD FPMOD.OBJ TO NEW.LIB
```

Removing Object Modules

The **DELETE** command removes object modules from a library file. This command must be entered in the following format:

```
DELETE libfile (modulename ⟦, modulename …⟧)
```

where

libfile is the name of an existing library file. The specified object modules are removed from this library.

modulename is the name of a module in the library file that you want to remove. Module names are entered in parentheses and are separated by commas.

The **DELETE** command may be entered on the command line or after the ***** prompt in interactive mode as shown in the following example.

```
LIB51 DELETE NEW.LIB (MODUL1)

* DELETE NEW.LIB (FPMULT, FPDIV)
```

Listing Library Contents

Use the **LIST** command to direct the LIB51 library manager to generate a listing of the object modules that are stored in a library file. **LIST** may be specified on the command line or after the ***** prompt in interactive mode. This command has the following format:

```
LIST libfile ⟦TO listfile⟧ ⟦PUBLICS⟧
```

where

libfile is the library file from which a module list is generated.

listfile is the file where listing information is written. If no ***listfile*** is specified, the listing information is displayed on the screen.

PUBLICS specifies that public symbols are included in the listing. Normally, only module names are listed.

Example:

```
LIB51 LIST NEW.LIB

* LIST NEW.LIB TO NEW.LST PUBLICS
```

The LIB51 library manager produces a module listing that appears as follows:

```
LIBRARY: NEW.LIB
   PUTCHAR
      _PUTCHAR
   PRINTF
      ?_PRINTF517?BYTE
      ?_SPRINTF517?BYTE
      ?_PRINTF?BYTE
      ?_SPRINTF?BYTE
      _PRINTF
      _SPRINTF
      _PRINTF517
      _SPRINTF517
   PUTS
      _PUTS
```

In this example, **PUTCHAR**, **PRINTF**, and **PUTS** are module names. The names listed below each of these module names are public symbols found in each of the modules.

Help Information

The **HELP** command directs the LIB51 library manager to display the available library manager commands. This command may be entered on the command line or at the ***** prompt in interactive mode. The LIB51 library manager responds with the following text:

```
ADD      {file[(module[,...])]} [,...] TO library_file
CREATE   library_file
DELETE   library_file(module[,...])
EXIT
HELP
LIST     library_file [TO file] [PUBLICS]
```

LIB51 Error Messages

This chapter lists the fatal and non-fatal errors that may be generated by the LIB51 library manager during execution. Each section includes a brief description of the message, as well as corrective actions you can take to eliminate the error or warning condition.

Fatal Errors

Fatal errors cause immediate termination of the LIB51 library manager. These errors normally occur as the result of a corrupt library or object file, or as a result of a specification problem involving library or object files.

Error	Error Message and Description
215	**CHECK SUM ERROR** **FILE: *filename*** The checksum for filename is incorrect. This usually indicates a corrupt file.
216	**INSUFFICIENT MEMORY** There is not enough memory for the LIB51 library manager to successfully complete the requested operation.
217	**NOT A LIBRARY** **FILE: filename** The filename that was specified is not a library file.
219	**NOT AN 8051 OBJECT FILE** **FILE: filename** The filename that was specified is not a valid 8051 object file.
222	**MODULE SPECIFIED MORE THAN ONCE** **MODULE: filename (modulename)** The specified modulename is included on the command line more than once.

Errors

The following errors cause immediate termination of the LIB51 library manager. These errors usually involve invalid command line syntax or I/O errors.

Error	Error Message and Description
201	**INVALID COMMAND LINE SYNTAX** A syntax error was detected in the command. The command line is displayed up to and including the point of error.
202	**INVALID COMMAND LINE, TOKEN TOO LONG** The command line contains a token that is too long for the LIB51 library manager to process.
203	**EXPECTED ITEM MISSING** The command line is incomplete. An expected item is missing.
205	**FILE ALREADY EXISTS** **FILE: filename** The filename that was specified already exists. This error is usually generated when attempting to create a library file that already exists. Erase the file or use a different filename.
208	**MISSING OR INVALID FILENAME** A filename is missing or invalid.
209	**UNRECOGNIZED COMMAND** A command is unrecognized by the LIB51 library manager. Make sure you correctly specified the command name.
210	**I/O ERROR ON INPUT FILE:** **system error message** **FILE: filename** An I/O error was detected when accessing one of the input files.
211	**I/O ERROR ON LIBRARY FILE:** **system error message** **FILE: filename** An I/O error was detected when accessing a library file.
212	**I/O ERROR ON LISTING FILE:** **system error message** **FILE: filename** An I/O error was detected when accessing a listing file.

Error	Error Message and Description
213	**I/O ERROR ON TEMPORARY FILE:** **system error message** **FILE: filename** An I/O error was detected when a temporary file was being accessed.
220	**INVALID INPUT MODULE** **FILE: filename** The specified input module is invalid. This error could be the result of an assembler error or could indicate that the input object file is corrupt.
221	**FILE SPECIFIED MORE THAN ONCE** **FILE: filename** The filename specified was included on the command line more than once.
223	**CANNOT FIND MODULE** **MODULE: filename (modulename)** The modulename specified on the command line was not located in the object or library file.
224	**ATTEMPT TO ADD DUPLICATE MODULE** **MODULE: filename (modulename)** The specified modulename already exists in the library file and cannot be added.
225	**ATTEMPT TO ADD DUPLICATE PUBLIC SYMBOL** **MODULE: filename (modulename)** **PUBLIC: symbolname** The specified public symbolname in modulename already exists in the library file and cannot be added.

Chapter 4. OC51 Banked Object File Converter

The OC51 Banked Object File Converter is an application that converts banked object files (object files created with the BL51 code banking linker/locator) into absolute object files.

The BL51 code banking linker/locator emits a special banked object file when it links a program that uses bank switching. Banked object files contain several banks of code that reside at the same physical location. For this reason, these object files are not compatible with Intel absolute OMF-51 object files. You must use the OC51 Banked Object File Converter to convert a single banked object file into one or more absolute object files.

The OC51 Banked Object File Converter will create an absolute object file for each code bank represented in the banked object file. Symbolic debugging information that was included in the banked object file will be copied to the absolute object modules that are generated.

Once you have used the OC51 Banked Object File Converter to create absolute object files, you may use the OH51 Object-Hex Converter to create Intel HEX files for each absolute object file.

The following sections describe how to use the OC51 Banked Object File Converter and list the errors that may be encountered during execution.

Using OC51

The OC51 Banked Object File Converter is invoked from the DOS prompt by typing **OC51** along with the name of the banked object file. The OC51 Banked Object File Converter command line must be entered according to the following format:

```
OC51 banked_obj_file
```

where

banked_obj_file is the name of the banked object file that is generated by the BL51 code banking linker/locator.

The OC51 Banked Object File Converter will create separate absolute object modules for each code bank represented in the banked object file. The absolute object modules will be created with a filename consisting of the *basename* of the banked object file combined with the file extension B*nn* where *nn* corresponds to the bank number 00-31. For example:

```
OC51 MYPROG
```

creates the absolute object files **MYPROG.B00** for code bank 0, **MYPROG.B01** for code bank 1, **MYPROG.B02** for code bank 2, etc.

NOTE

You should use the OC51 Banked Object File Converter only if you used the ***BANKx****,* ***BANKAREA****, or* ***COMMON*** *directives on the BL51 code banking linker/locator command line to specify code banking is in effect.*

If your program does not use code banking, do not use the OC51 Banked Object File Converter to generate an absolute object module (even if you linked using the BL51 code banking linker/locator).

The OC51 Banked Object File Converter may simultaneously open as many as 17 files. You should verify that the FILES statement in your **CONFIG.SYS** *file specifies more than 17 files. Refer to your DOS manual for more information.*

OC51 Error Messages

This chapter lists the errors that you may encounter when you use the OC51 Banked Object File Converter. Each message includes a brief description of the message as well as corrective actions you can take to eliminate the error condition.

Fatal Errors

Error	Error Message and Description
201	**FILE ACCESS ERROR ON INPUT FILE** **FILE: *filename*** An error occurred while reading the specified file.
202	**FILE ACCESS ERROR ON OUTPUT FILE** **FILE: *filename*** An error occurred while writing the specified file.
203	**NOT A BANKED 8051 OBJECT FILE** The input file is not a banked object file.
204	**INVALID INPUT FILE** The input file has an invalid format.
205	**CHECKSUM ERROR** The input file has an invalid checksum. This error is usually caused by an error from the BL51 code banking linker/locator. Make sure that your program was linked successfully.
206	**INTERNAL ERROR** The OC51 Banked Object File Converter has detected an internal error. Contact technical support.
207	**SCOPE LEVEL ERROR** **MODULE: *modulename*** The symbolic information in the specified file contains errors. This error message is usually the result of an error at link time. Make sure that your program was linked successfully.

Error	Error Message and Description
208	**PATH OR FILE NOT FOUND** **FILE: *filename*** The OC51 Banked Object File Converter cannot find the specified file. Make sure the file actually exists.

Chapter 5. OH51 Object-Hex Converter

OH51 is an application that converts absolute object files into Intel HEX files.

Program code stored in the absolute object file is converted into hexadecimal values and is output to a file in Intel HEX file format. The Intel HEX file may then be used by an EPROM programmer or emulator.

The following sections describe how to use the OH51 program, the command-line options that are available, and any errors that may be encountered during execution.

Using OH51

To invoke OH51 from the DOS prompt, type **OH51** along with the name of the absolute object file. The OH51 command line must be entered in the following format:

```
OH51 absolute_obj_file ⟦HEXFILE (filename)⟧
```

where

absolute_obj_file is the name of the absolute object file that is generated by the L51 linker/locator.

filename is the name of the Intel HEX file to generate. By default, the name given to the HEX file is the base name of the ***absolute_obj_file*** followed by the **.HEX** extension.

OH51 Error Messages

This chapter lists fatal error, syntax error, and warning messages that you may encounter when using OH51. Each section includes a brief description of the message as well as corrective actions you can take to eliminate the error or warning condition.

Fatal Errors

Fatal errors cause immediate termination of the object file conversion. These errors normally occur as the result of a corrupt absolute object file.

`*** FATAL ERROR: INVALID RECORD-TYPE ENCOUNTERED`
The absolute object file contains an invalid record type.

`*** FATAL ERROR: INCONSISTENT OBJECT FILE`
The input file has an invalid format.

Errors

The following errors cause immediate termination of the object file conversion. They normally occur as the result of invalid or incomplete options specified on the command line.

`*** ERROR, ARGUMENT TOO LONG`
An argument in the command line is too long.

`*** ERROR, DELIMITER '(' AFTER PARAMETER EXPECTED`
The command-line parameter must be followed by an argument enclosed in parentheses ().

`*** ERROR, DELIMITER ')' AFTER PARAMETER EXPECTED`
The command-line parameter must be followed by an argument enclosed in parentheses ().

`*** ERROR, UNKNOWN CONTROL:`
The specified command-line parameter is unrecognized.

`*** ERROR, RESPECIFIED CONTROL, IGNORED`
The indicated command-line control was specified twice.

Warnings

Warnings signal that a problem was encountered during the object file conversion process, but the generated hex file may still be valid. Warnings do not hinder the object file conversion.

WARNING: <PUBDEF> HEX-FILE WILL BE INVALID

The absolute object file still contains public definitions. This warning usually indicates that the object file has not been processed by the L51 linker/locator. The hex file that is produced may be invalid.

WARNING: <EXTDEF> UNDEFINED EXTERNAL

The absolute object file still contains external definitions. This warning usually indicates that the object file has not been processed by the L51 linker/locator. The hex file that is produced may be invalid.

Intel HEX File Format

The Intel HEX file is an ASCII text file with lines of text that follow the Intel HEX file format. Each line in an Intel HEX file contains one HEX record. These records are made up of hexadecimal numbers that represent machine language code and/or constant data. Intel HEX files are often used to transfer the program and data that would be stored in a ROM or EPROM. Most EPROM programmers or emulators can use Intel HEX files.

Record Format

An Intel HEX file is composed of any number of HEX records. Each record is made up of five fields that are arranged in the following format:

```
:llaaaatt[dd...]cc
```

Each group of letters corresponds to a different field, and each letter represents a single hexadecimal digit. Each field is composed of at least two hexadecimal digits—which make up a byte—as described below:

: is the colon that starts every Intel HEX record.

ll is the record-length field that represents the number of data bytes (**dd**) in the record.

aaaa is the address field that represents the starting address for subsequent data in the record.

tt is the field that represents the HEX record type, which may be one of the following:

00	data record
01	end-of-file record

dd is a data field that represents one byte of data. A record may have multiple data bytes. The number of data bytes in the record must match the number specified by the **ll** field.

cc is the checksum field that represents the checksum of the record. The checksum is calculated by summing the values of all hexadecimal digit pairs in the record modulo 256 and taking the two's complement.

Data Records

The Intel HEX file is made up of any number of data records that are terminated with a carriage return and a linefeed. Data records appear as follows:

```
:10246200464C5549442050524F46494C4500464C33
```

where:

10 is the number of data bytes in the record.

2462 is the address where the data are to be located in memory.

00 is the record type 00 (a data record).

464C...464C is the data.

33 is the checksum of the record.

End-of-File (EOF) Records

An Intel HEX file must end with an end-of-file (EOF) record. This record must have the value 01 in the record type field. An EOF record always appears as follows:

```
:00000001FF
```

where:

00 is the number of data bytes in the record.

0000 is the address where the data are to be located in memory. The address in end-of-file records is meaningless and is ignored. An address of 0000h is typical.

01 is the record type 01 (an end-of-file record).

FF is the checksum of the record and is calculated as **`01h + NOT(00h + 00h + 00h + 01h)`**.

Example Intel HEX File

Following is an example of a complete Intel HEX file:

```
:10001300AC12AD13AE10AF1112002F8E0E8F0F2244
:10000300E50B250DF509E50A350CF5081200132259
:03000000020023D8
:0C002300787FE4F6D8FD7581130200031D
:10002F00EFF88DF0A4FFEDC5F0CEA42EFEEC88F016
:04003F00A42EFE22CB
:00000001FF
```

Glossary

A51
The command used to assemble programs using the A51 Macro Assembler.

aggregate types
Arrays, structures, and unions.

argument
The value that is passed to macro or function.

arithmetic types
Data types that are integral, floating-point, or enumerations.

array
A set of elements all of the same data type.

ASCII
American Standard Code for Information Interchange. This is a set of 256 codes used by computers to represent digits, characters, punctuation, and other special symbols. The first 128 characters are standardized. The remaining 128 are defined by the implementation.

basename
The part of the file name that excludes the drive letter, directory name, and file extension. For example, the basename for the file **`C:\SAMPLE\SIO.A51`** is **`SIO`**.

batch file
A text file that contains MS-DOS commands and programs that can be invoked from the command line.

BL51
The command used to link object files and libraries using the 8051 Code Banking Linker/Locator.

C51
The command used to compile programs using the 8051 Optimizing C Cross Compiler.

code banking
See bank switching.

constant expression
Any expression that evaluates to a constant non-variable value. Constants

may include character, integer, enumeration, and floating-point constant values.

declaration
A C construct that associates the attributes of a variable, type, or function with a name.

definition
A C construct that specifies the name, formal parameters, body, and return type of a function or that initializes and allocates storage for a variable.

directive
An instruction to the C preprocessor or a control switch to the C51 compiler.

DS51
The command used to load and execute the DS51 Debugger/Simulator.

environment table
The memory area used by MS-DOS to store environment variables and their values.

environment variable
A variable stored in the environment table. These variables provide MS-DOS programs with information like where to find include files and library files.

escape sequence
A backslash (‘\’) character followed by a single letter or a combination of digits that specifies a particular character value in strings and character constants.

expression
A combination of any number of operators and operands that produces a constant value.

function
A combination of declarations and statements that can be called by name that perform an operation and/or return a value.

function call
An expression that invokes and possibly passes arguments to a function.

in-circuit emulator (ICE)
A hardware device that aids in debugging embedded software by providing hardware-level single-steping, tracing, and break-pointing. Some ICEs provide a trace buffer that stores the most recent CPU events.

include file
A text file that is incorporated into a source file using the **#include** preprocessor directive.

keyword
A reserved word with a predefined meaning for the compiler.

L51
The command used to link object files and libraries using the 8051 Linker/Locator.

LIB51
The command used to manipulate 8051 library files using the 8051 Library Manager.

library
A file that stores a number of possibly related object modules. The linker can extract modules from the library to use in building a target object file.

macro
An identifier that represents a series of keystrokes that is defined using the **#define** preprocessor directive.

manifest constant
A macro that is defined to have a constant value.

MCS-51
The general name applied to the entire family of 8051 compatible microprocessors.

memory model
Any of the models that specifies which memory areas are used for function arguments and local variables.

monitor51
An 8051 program that can be loaded into your target CPU to aid in debugging and rapid product development through rapid software downloading.

newline character
The character used to mark the end of a line in a text file or the escape sequence (**‘\n’**) used to represent the newline character.

null character
The ASCII character with the value 0 represented as the escape sequence (**‘\0’**).

null pointer
A pointer that references nothing and has an offset of 0000h. A null pointer has the integer value 0.

object
An area of memory that can be examined. Usually used when referring to the memory area associated with a variable or function.

object file
A file, created by the compiler, that contains the program segment information and relocatable machine code.

OH51
The command used to convert absolute object files into other hexadecimal file formats using the Object File Converter.

operand
A variable or constant that is used in an expression.

operator
A symbol that specifies how to manipulate the operands of an expression; e.g., +, -, *, /.

parameter
The value that is passed to a macro or function.

PL/M-51
A high-level programming language that provides a blocked structure, a facility for data structures, type checking, and a standard language for use on most Intel hardware architectures.

pointers
A variable that contains the address of another variable, function, or memory area.

pragma
A statement that passes an instruction to the compiler at compile time.

relocatable
Able to be moved or relocated. Not containing absolute or fixed addresses.

RTX51 Full
An 8051 Real-Time Executive that provides a multitasking operating system kernel and library of routines for its use.

RTX51 Tiny
A limited version of RTX51.

scalar types
Integer, enumerated, floating-point, and pointer types.

scope
The sections or a program where an item (function or variable) can be referenced by name. The scope of an item may be limited to file, function, or block.

source file
A text file containing assembly program code.

stack
An area of memory, indirectly accessed by a stack pointer, that shrinks and expands dynamically as items are pushed onto the stack and popped off of the stack. Items in the stack are removed on a LIFO (last-in, first-out) basis.

static
A storage class that, when used with a variable declaration in a function, causes variables to retain their value after exiting the block or function in which they are declared.

string
An array of characters that is terminated with a null character (**'\0'**).

string literal
A string of characters enclosed within double quotes (**" "**).

token
A fundamental symbol that represents a name or entity in a programming language.

TS51
The command used to load and execute the 8051 TS51 Target Debugger.

two's complement
A binary notation that is used to represent both positive and negative numbers. Negative values are created by complementing all bits of a positive value and adding 1.

type
A description of the range of values associated with a variable. For example, an **int** type can have any value within its specified range (-32768 to 32767).

whitespace character
Characters that are used as delimiters in C programs such as space, tab, newline, etc.

wild card

One of the MS-DOS characters (? or *) that can be used in place of characters in a filename.

Index

A

A51, defined....155
Absolute Address Calculation....8
Absolute Object File....8
Absolute object files....3
ADD....136
Additional items, notational conventions....iv
aggregate types, defined....155
argument, defined....155
arithmetic types, defined....155
array, defined....155
ASCII, defined....155
Automatic Bank Selection....49

B

BA....15
Bank Switching....10
Bank Switching Configuration....50
Bank Switching Directives....42
BANKAREA....15,45,57
Banked object files....3
Banking With Four 64 KByte Banks....53
BANKx....15,45,58
basename, defined....155
batch file, defined....155
BI....15,59
BIT....15,25,59
BL51....1
BL51 code banking linker/locator....1
BL51 Error Messages....88
BL51, defined....155
bold capital text, use of....iv
braces, use of....iv
Bx....15

C

C51, defined....155
C51LIB....33
Choices, notational conventions....iv
CO....15,60
CODE....15,30,60
Code Bank Areas....44
code banking, defined....155
Code Space....7
Combining Program Modules....3
Combining Segments....4
Command Files....13
Command-Line Directives....14
Command-Line Examples....13
COMMON....15,46,61
Common Code Area....42
constant expression, defined....155
courier typeface, use of....iv
CREATE....136

D

DA....15,62
DATA....15,26,62
declaration, defined....156
definition, defined....156
DELETE....136
Directive Reference....56
Directive Summary....16
directive, defined....156
Displayed text, notational conventions....iv
Document conventions....iv
double brackets, use of....iv
DS51, defined....156

E

ellipses, use of....iv
ellipses, vertical, use of....iv
environment table, defined....156

environment variable, defined.......... 156
EOF records.................................... 153
Errorlevel.. 14
escape sequence, defined 156
Exceptions 97
EXIT... 136
expression, defined 156
External Data Space............................ 7

F

Filename, notational conventions........ iv
function call, defined 156
function, defined 156

G

Generating a Listing File...................... 9
Generating an Absolute Object
File 8

H

HELP 136
High-Level Language Directives 32

I

ICE, defined..................................... 156
ID.. 15,63
IDATA.................................. 15,27,63
in-circuit emulator, defined............. 156
include file, defined 157
Intel HEX
- Data records 153
- End-of-file records 153
- EOF records 153
- Example file 154
- Record format 152

Intel HEX file format....................... 152
Intel HEX files.............................. 3,149
Internal Data Space.............................. 6
italicized text, use of iv
IX.. 15,64
IXREF..................................... 15,17,64

K

Key names, notational
conventions.......................................iv
keyword, defined..............................157

L

L51, defined 157
LIB51 135
- Adding Object Modules 138
- Command line.............................135
- Commands 136
- Creating a Library.......................137
- Error Messages 142
- Help Information 141
- Interactive mode 135
- Listing Library Contents 140
- Removing Object Modules 139

LIB51, defined 157
Library Manager...............................135
library, defined 157
Linking Programs...............................11
LIST 136
Listing File 9
Listing File Directives........................17
Locating Segments 6
Long Command Lines 12

M

macro, defined..................................157
manifest constant, defined................157
MCS-51, defined 157
memory model, defined....................157
monitor51, defined 157

N

NA..15,65
NAME.....................................15,22,65
newline character, defined................157
NLIB 15,70
NOAMAKE 15,22,66
NODEBUGLINES...................15,22,67
NODEBUGPUBLICS..............15,23,68
NODEBUGSYMBOLS 15,24,69

NODEFAULTLIBRARY.........15,32,70
NODL..15,67
NODP..15,68
NODS..15,69
NOGENERATED 18
NOGN 18
NOLI 15,18,71
NOLIBRARIES.................................18
NOLINES................................15,19,71
NOMA...15,72
NOMAP 15,18,72
Non-Fatal Errors.................................91
NOOL..15,75
NOOVERLAY 15,33,75
NOPU..15,73
NOPUBLICS............................15,18,73
NOSY 15,74
NOSYMBOLS 15,18,74
null character, defined 157
null pointer, defined..........................158

O

object file, defined............................158
object, defined 158
OC51 3,145
Command line.............................145
OH51 3,149
Command line.............................149
Error messages............................150
OH51, defined 158
OL..15,75
Omitted text, notational conventions......................................iv
operand, defined 158
operator, defined...............................158
Optimum Program Structure with Bank Switching........................44
Optional items, notational conventions......................................iv
Ordering Segments in a Bank 47
Output File..14
OVERLAY 15,34,75
Overlaying Data Memory 7

P

PAGELENGTH 15,17,77
PAGEWIDTH.......................... 15,17,78
parameter, defined............................ 158
PC 16,80
PDATA 16,32,79
PL.. 15,77
PL/M-51, defined 158
pointers, defined.............................. 158
PR 16,81
pragma, defined............................... 158
PRECEDE............................... 16,28,80
PRINT.................................... 16,17,81
Printed text, notational conventions iv
Public Symbols in L51_BANK.A51 52
PW 15,78

R

RAMSIZE 16,25,82
REGFILE 16,42,83
relocatable, defined 158
Resolving External References 8
RS 16,82
RTX51 11,16,49,84
RTX51 and RTX51 Tiny Directives 49
RTX51 Tiny...................................... 11
RTX51 Tiny, defined 158
RTX51, defined 158
RTX51TINY 16,49,85

S

sans serif typeface, use of.................... iv
scalar types, defined......................... 159
scope, defined 159
Segment Naming Conventions 4
source file, defined........................... 159
Specifying Code Banks and Common Code Areas 44
ST.. 16,86
STACK 16,29,86

stack, defined 159
static, defined.................................. 159
string literal, defined........................ 159
string, defined 159

T

token, defined 159
TS51, defined.................................. 159
two’s complement, defined 159
type, defined 159

V

Variables, notational
  conventions iv
vertical bar, use of...............................iv

W

Warnings 88
whitespace character, defined...........159
wild card, defined.............................160

X

XD...16,87
XDATA....................................16,31,87