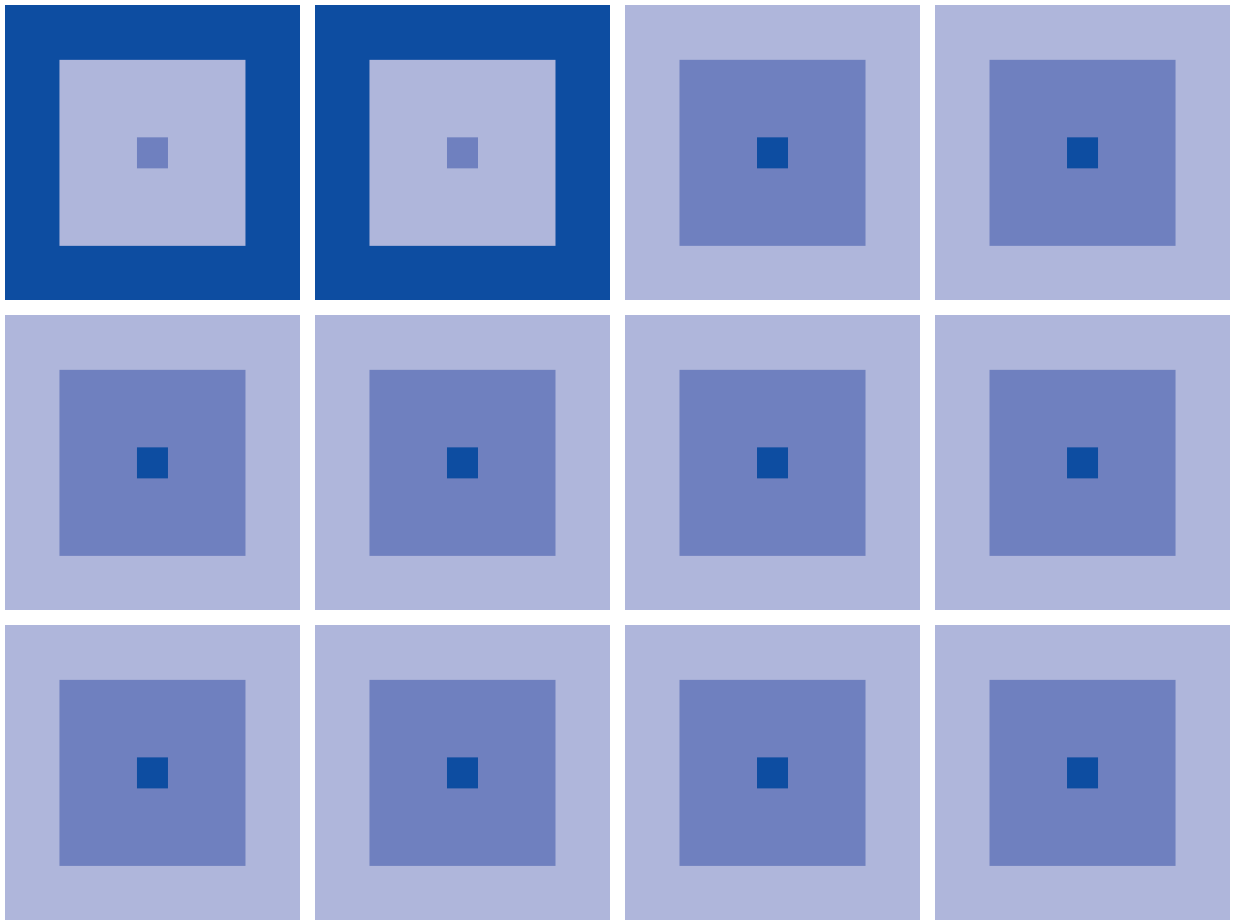
EPSON
EXCEED YOUR VISION

CMOS 8-BIT SINGLE CHIP MICROCOMPUTER

S5U1C88000C Manual I

(Integrated Tool Package for S1C88 Family)

C Compiler/Assembler/Linker



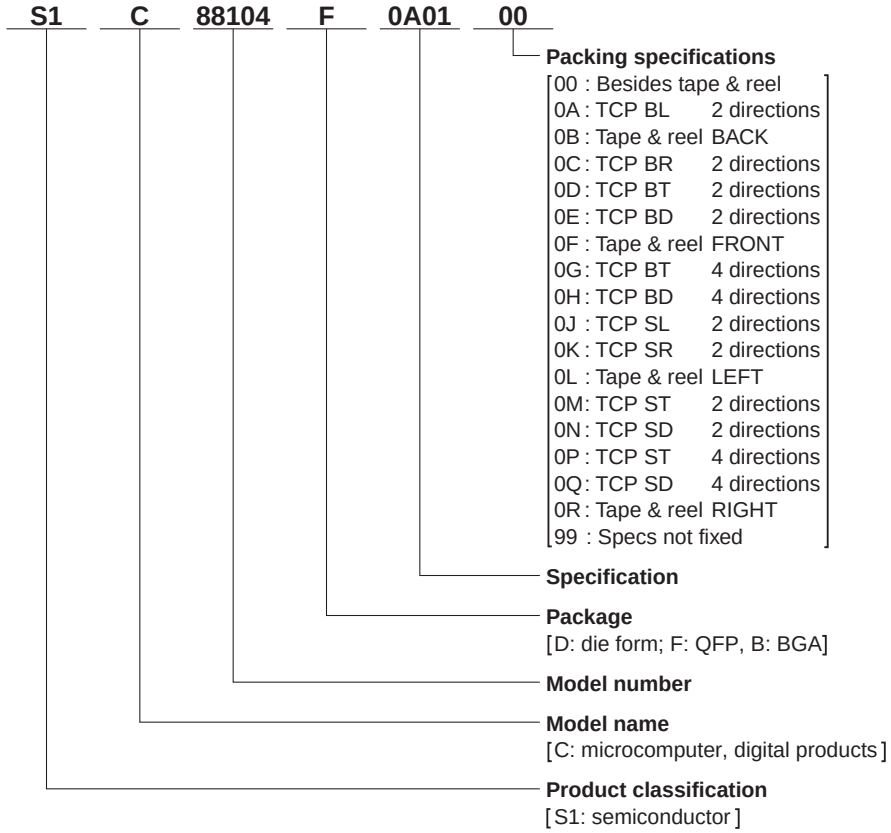
SEIKO EPSON CORPORATION

NOTICE

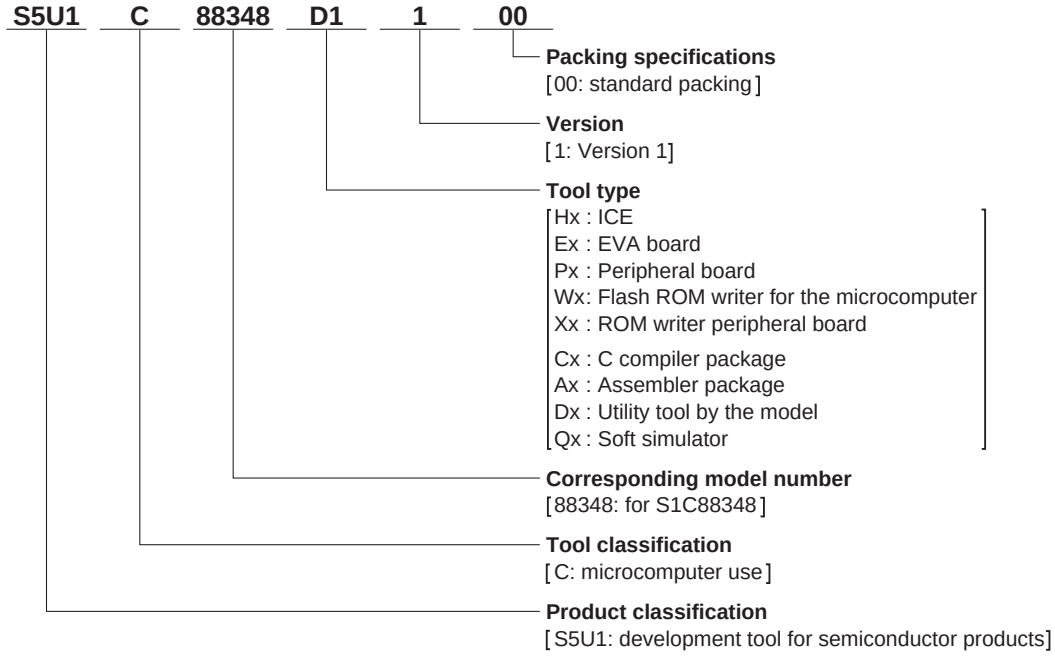
No part of this material may be reproduced or duplicated in any form or by any means without the written permission of Seiko Epson. Seiko Epson reserves the right to make changes to this material without notice. Seiko Epson does not assume any liability of any kind arising out of any inaccuracies contained in this material or due to its application or use in any product or circuit and, further, there is no representation that this material is applicable to products requiring high level reliability, such as medical products. Moreover, no license to any intellectual property rights is granted by implication or otherwise, and there is no representation or warranty that anything made in accordance with this material will be free from any patent or copyright infringement of a third party. This material or portions thereof may contain technology or the subject relating to strategic products under the control of the Foreign Exchange and Foreign Trade Law of Japan and may require an export license from the Ministry of Economy, Trade and Industry or other approval from another government agency.

Configuration of product number

Devices



Development tools



MANUAL ORGANIZATION

The S1C88 Family Integrated Tool Package contains the tools required to develop software for the S1C88 Family microcomputers. The S5U1C88000C Manual (S1C88 Family Integrated Tool Package) describes the tool functions and how to use the tools. The manual is organized into two documents as shown below.

I. C Compiler/Assembler/Linker (this document)

Describes the C Compiler and its tool chain ([Main Tool Chain][note] part shown in the figure on the next page).

II. Workbench/Development Tools/Assembler Package Old Version

Describes the Work Bench that provides an integrated development environment, Advanced Locator, the Mask Data Creation Tools ([Development Tool Chain] part shown in the figure on the next page), Debugger, and Structured Assembler ([Sub Tool Chain] part shown in the figure on the next page).

This manual assumes that the reader is familiar with C and Assembly languages.

Refer to the following manuals as necessary when developing an S1C88xxx microcomputer:

S1C88xxx Technical Manual
: Describes the device specifications, control method and Flash EEPROM programming.

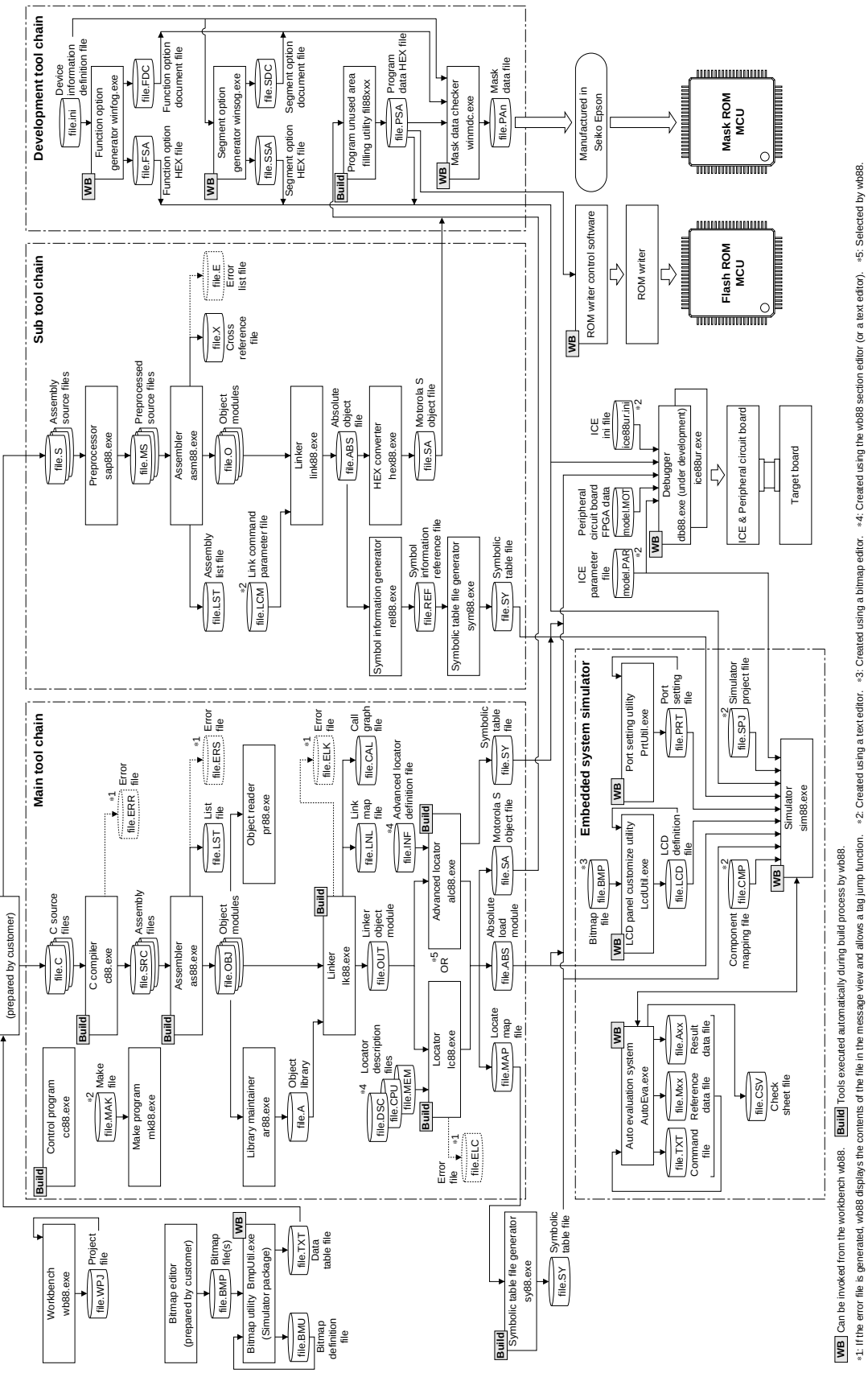
S5U1C88000Q Manual
: Describes the operation of the tools included in the Simulator Package.

S5U1C88000H5 Manual
: Describes the operation of the ICE (S5U1C88000H5).

S5U1C88xxxP Manual
: Describes the operation of the peripheral circuit board installed in the ICE.

Note: [Main Tool Chain] has added Advanced Locator alc88 beginning with Ver. 3, which can be used in place of Locator lc88. This manual is written for the TASKING tool chain, and only describes the C Compiler, Assembler, Linker, Locator, and related information. For details about Advanced Locator not included in the TASKING tool chain, refer to document II, "Workbench/Development Tools/ Assembler Package Old Version".
Except when using existing resources including locator description files, you need not learn a description language for relocating objects, which is why we recommend using Advanced Locator with a branching optimization function. Therefore, when developing new S1C88 Family applications (i.e., when using Advanced Locator), you need not specifically read the description of Locator lc88 in Chapter 4 and DELFEE (the locator description language) in Chapter 5 of this manual. The locator functions and operations described elsewhere in this manual are to be replaced by Advanced Locator.

S1C88 FAMILY INTEGRATED DEVELOPMENT ENVIRONMENT



WB Can be invoked from the workbench wb88. Build Tools executed automatically during build process by wb88.

*1: If the error file is generated, wb88 displays the contents of the file in the message view and allows a tag jump function. *2: Created using a text editor. *3: Created using a bitmap editor. *4: Created using the wb88 section editor (or a text editor). *5: Selected by wb88.

CONTENTS

CHAPTER 1 C COMPILER

1.1 Overview

1.1.1 Introduction to S1C88 C Cross-Compiler

This manual provides a functional description of the S1C88 C Cross-Compiler. This manual uses **c88** (the name of the binary) as a shorthand notation for "S1C88 C Compiler".

SEIKO EPSON offers a complete tool chain for the S1C88 family of processors. 'S1C88' is used as a shorthand notation for the S1C88 family of processors and their derivatives.

The S1C88 C compiler accepts source programs written in ANSI C and translates these into S1C88 assembly source code files. The S1C88 C cross-compiler generates code for the S1C88 operating in 'native' mode. The compiler accepts language extensions to improve code performance and to allow the use of typical S1C88 architectural provisions efficiently at the C level. The compiler is ANSI C compatible and consists of three major parts; the *preprocessor*, the S1C88 C *frontend* and the associated *backend* or code generator. These are all integrated into a single program to avoid the need of intermediate files, thus speeding up the compilation process. It also simplifies the implementation of joint frontend-backend optimization strategies and preprocessor pragmas. This effectively makes the compiler a one pass compiler, with minimum file I/O overhead.

The compiler processes one C function at a time, until the entire source module has been read. The function is parsed, checked on semantic correctness and then transformed into an intermediate code tree that is stored in memory. Code optimizations are performed during the construction of the intermediate code, and are also applied when the complete function has been processed. The latter are often referred to as *global* optimizations.

c88 generates assembly source code using the S1C88 assembly language specification, you must assemble this code with the S1C88 Cross-Assembler. This manual uses **as88** as a shorthand notation for "S1C88 Cross-Assembler".

You can link the generated object with other objects and libraries using the **lk88** S1C88 linker. In this manual we use **lk88** as a shorthand notation for "**lk88** S1C88 linker". You can locate the linked object to a complete application using the **lc88** S1C88 locator. In this manual we use **lc88** as a shorthand notation for "**lc88** S1C88 locator".

The program **cc88** is a control program. The control program facilitates the invocation of various components of the S1C88 tool chain. **cc88** recognizes several filename extensions. C source files (`.c`) are passed to the compiler. Assembly sources (`.asm`) are preprocessed and passed to the assembler. Relocatable object files (`.obj`) and libraries (`.a`) are recognized as linker input files. Files with extension `.out` and `.dsc` are treated as locator input files. The control program supports options to stop at any stage in the compilation process and has options to produce and retain intermediate files.

1.1.2 General Implementation

This section describes the different phases of the compiler and the target independent optimizations.

1.1.2.1 Compiler Phases

During the compilation of a C program, a number of phases can be identified. These phases are divided into two groups, referred to as *frontend* and *backend*.

frontend:

The preprocessor phase:

File inclusion and macro substitution are done by the preprocessor before parsing of the C program starts. The syntax of the macro preprocessor is independent of the C syntax, but also described in the ANSI X3.159-1989 standard.

The scanner phase:

The scanner converts the preprocessor output to a stream of tokens.

The parser phase:

The tokens are fed to a parser for the C grammar. The parser performs a syntactic and semantic analysis of the program, and generates an intermediate representation of the program.

The frontend optimization phase:

Target processor independent optimization is performed by transforming the intermediate code. The next section discusses the frontend optimizations.

backend:

The backend optimization phase:

Performs target processor specific optimizations. Very often this means another transformation of the intermediate code and actions like register allocation techniques for variables, expression evaluation and the best usage of the addressing modes. Section 1.2, "Language Implementation", discusses this item in more detail.

The code generator phase:

This phase converts the intermediate code to an internal instruction code, representing the S1C88 assembly instructions.

The peephole optimizer / pipeline scheduler phase:

This phase uses pattern matching techniques to perform peephole optimizations on the internal code. The pipeline scheduler reorders and combines instructions to minimize the number of instructions. Finally the peephole optimizer translates the internal instruction code into assembly code for **as88**. The generated assembly does not contain any macros. The assembler is also equipped with an optimizer.

All phases (of both frontend and backend) of the compiler are combined into one program. The compiler does not use intermediate files for communication between the different phases of compilation. The backend part is not called for each C statement, but starts after a complete C function has been processed by the frontend (in memory), thus allowing more optimization. The compiler only requires one pass over the input file, resulting in relatively fast compilation.

1.1.2.2 Frontend Optimizations

The command line option **-O** controls the amount of optimization applied on the C source. Within a source file, the pragma `#pragma optimize` sets the optimization level of the compiler. Using the pragma, certain optimizations can be switched on or off for a particular part of the program. Several optimizations cannot be controlled individually. e.g., constant folding will always be done.

The compiler performs the following optimizations on the intermediate code. They are independent of the target processor and the code generation strategy:

Constant folding
: Expressions only involving constants are replaced by their result.

Expression rearrangement
: Expressions are rearranged to allow more constant folding.
E.g. `1+ (x-3)` is transformed into `x + (1-3)`, which can be folded.

Expression simplification
: Multiplication by 0 or 1 and additions or subtractions of 0 are removed. Such useless expressions may be introduced by macros, or by the compiler itself (e.g., array subscription).

Logical expression optimization
: Expressions involving '&&', '||' and '!' are interpreted and translated into a series of conditional jumps.

Loop rotation
: With `for` and `while` loops, the expression is evaluated once at the 'top' and then at the 'bottom' of the loop. This optimization does not save code, but speeds up execution.

Switch optimization
: A number of optimizations of a switch statement are performed, such as the deletion of redundant case labels or even the deletion of the switch.

Control flow optimization
: By reversing jump conditions and moving code, the number of jump instructions is minimized. This reduces both the code size and the execution time.

Jump chaining
: A conditional or unconditional jump to a label which is immediately followed by an unconditional jump may be replaced by a jump to the destination label of the second jump. This optimization does not save code, but speeds up execution.

Remove useless jumps
: An unconditional jump to a label directly following the jump is removed. A conditional jump to such a label is replaced by an evaluation of the jump condition. The evaluation is necessary because it may have side effects.

Conditional jump reversal
: A conditional jump over an unconditional jump is transformed into one conditional jump with the jump condition reversed. This reduces both the code size and the execution time.

Cross jumping and branch tail merging
: Identical code sequences in two different execution paths are merged when this is possible without adding extra instructions. This transformation decreases code size rather than execution time, but under certain circumstances it avoids the execution of one jump.

Constant/copy propagation
: A reference to a variable with known contents is replaced by those contents.

Common subexpression elimination
: The compiler has the ability to detect repeated uses of the same (sub-) expression. Such a "common" expression may be temporarily saved to avoid recomputation. This method is called *common subexpression elimination*, abbreviated CSE.

Dead code elimination
: Unreachable code can be removed from the intermediate code without affecting the program. However, the compiler generates a warning message, because the unreachable code may be the result of a coding error.

Loop optimization
: Invariant expressions may be moved out of a loop and expressions involving an index variable may be reduced in strength.

Loop unrolling
: Eliminate short loops by replacing them with a number of copies.

1.1.2.3 Backend Optimizations

The following optimizations are target dependent and are therefore performed by the backend.

Allocation graph
: Variables, parameters, intermediate results and common subexpressions are represented in allocation units. Per function, the compiler builds a graph of allocation units which indicates which units are needed and when. This allows the register allocator to get the most efficient occupation of the available registers. The compiler uses the allocation graph to generate the assembly code.

Peephole optimizations
: The generated assembly code is improved by replacing instruction sequences by equivalent but faster and/or shorter sequences, or by deleting unnecessary instructions.

Leaf function handling
: Leaf functions (function not calling other functions), are handled specially with respect to stack frame building.

Dead store elimination
: Expressions from which the result is never used are eliminated.

Tail recursion elimination
: Replace a recursion statement to branch to the beginning of the statement.

1.1.3 Compiler Structure

If you want to build an S1C88 application you need to invoke the following programs directly, or via the control program:

- The C compiler (**c88**), which generates an assembly source file from the file with suffix `.c`. The suffix of the compiler output file is `.src`. However, you can direct the output to another file with the **-o** option. C source lines can be intermixed with the generated assembly statements with the **-s** option. High level language debugging information can be generated with the **-g** option. You are advised not to use the **-g** option when inspecting the generated assembly source code, because it contains a lot of 'unreadable' high level language debug directives. The C compilers make only one pass on every file. This pass checks the syntax, generates the code and performs code optimization.
- The corresponding cross-assembler (**as88**), which processes the generated assembly source file into a relocatable object file with suffix `.obj`.
- The **lk88** linker, which links the generated relocatable object files and C-libraries. The result is a relocatable object file with suffix `.out`. A linker map file with suffix `.lnl` is available after this stage.
- The **lc88** locator, which locates the generated relocatable object files. The result is an absolute loadable file with suffix `.abs`. A full application map file with suffix `.map` is available after this stage.

You can directly load the output file of the locator with extension `.abs` into the debugger.

The next figure explains the relationship between the different parts of the S1C88 tool chain:

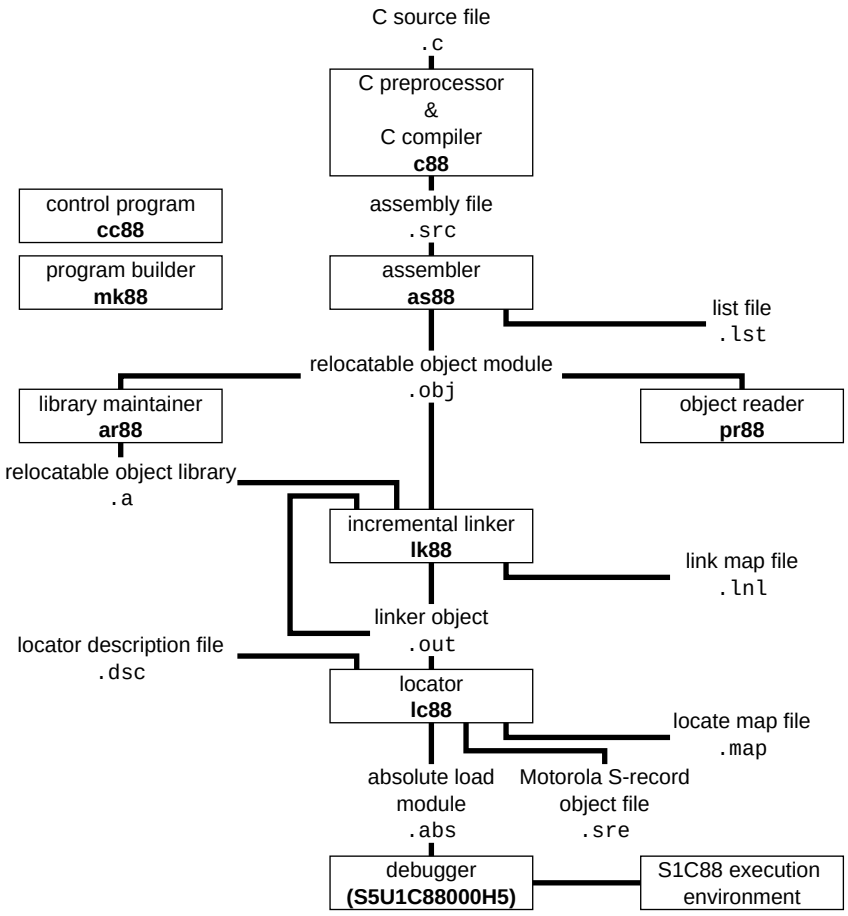


Fig. 1.1.3.1 S1C88 development flow

The program **cc88** is a so-called control program, which facilitates the invocation of various components of the S1C88 tool chain. C source programs are compiled by the compiler, assembly source files are passed to the assembler. A C preprocessor program is available as an integrated part of the C compiler. The control program recognizes the file extensions `.a` and `.obj` as input files for the linker. The control program passes files with extensions `.out` and `.dsc` to the locator. All other files are considered to be object files and are passed to the linker. The control program has options to suppress the locating stage (**-cl**), the linker stage (**-c**) or the assembler stage (**-cs**).

Optionally the locator, **lc88** produces output files in Motorola S-record format. The default output format is IEEE-695.

Normally, the control program removes intermediate compilation results, as soon as the next phase completes successfully. If you want to retain all intermediate files, the option **-tmp** prevents removal of these files.

For a description of all utilities available and the possible output formats of the locator, see the respective sections.

1.1.4 Environment Variables

This section contains an overview of the environment variables used by the S1C88 tool chain.

Environment Variable	Description
AS88INC	Specifies an alternative path for include files for the assembler.
C88INC	Specifies an alternative path for #include files for the C compiler **c88**.
C88LIB	Specifies a path to search for library files used by the linker **lk88**.
CC88BIN	When this variable is set, the control program, **cc88**, prepends the directory specified by this variable to the names of the tools invoked.
CC88OPT	Specifies extra options and/or arguments to each invocation of **cc88**. The control program processes the arguments from this variable before the command line arguments.
PATH	Specifies the search path for your executables.
TMPDIR	Specifies an alternative directory where programs can create temporary files. Used by **c88**, **cc88**, **as88**, **lk88**, **lc88**, **ar88**.

1.1.4.1 Using the Control Program

A detailed description of the process using the sample program `calc.c` is described below. This procedure is outlined as a guide for you to build your own executables for debugging.

1. Make the subdirectory `c` of the `examples` directory the current working directory.
2. Be sure that the directory of the binaries is present in the PATH environment variable.
3. Compile, assemble, link and locate the modules using one call to the control program **cc88**:

   ```
   cc88 -g -M -Ml -calc.c -o calc.abs
   ```

   The **-g** option specifies to generate symbolic debugging information. This option must always be specified when debugging with the debugger.
   Some optimizations may affect the ability to debug the code in a high level language debugger. Therefore, the **-O0** option must be selected with **-g** to switch off these optimizations. When the **-g** option is specified to the compiler with a higher optimization level, the compiler will issue warning message W555.
   The **-M** option specifies to generate map files.
   The **-Ml** option specifies to use the large memory model.
   The **-o** option specifies the name of the output file.

The command in step 3 generates the object file `calc.obj`, the linker map file `calc.lnl`, the locator map file `calc.map` and the absolute output file `calc.abs`. The file `calc.abs` is in the IEEE Std. 695 format, and can directly be used by the debugger. No separate formatter is needed.

Now you have created all the files necessary for debugging with the debugger with one call to the control program.

If you want to see how the control program calls the compiler, assembler, linker and locator, you can use the **-v0** option or **-v** option. The **-v0** option only displays the invocations without executing them. The **-v** option also executes them.

```
cc88 -g -M -Ml calc.c -o calc.abs -v0
```

The control program shows the following command invocations without executing them:

```
S1C88 control program va.b rc        SN00000000-003 (c) year TASKING, Inc.
+ c88 -e -g -Ml -o /tmp/cc24611b.src calc.c
+ as88 -e -g -o calc.obj /tmp/cc24611b.src
+ lk88 -e -M calc.obj -lcl -lrt -lfp -ocalc.out -Ocalc
+ lc88 -e -M -ocalc.abs calc.out
```

The **-e** option removes output files after errors occur. The **-O** option of the linker specifies the basename of the map file. The **-lcl**, **-lrt** and **-lfp** options of the linker specify to link the appropriate C library, run-time library and floating point library.

As you can see, the tools use temporary files for intermediate results. If you want to keep the intermediate files you can use the **-tmp** option. The following command makes this clear.

```
cc88 -g -M -Ml calc.c -o calc.abs -v0 -tmp
```

This command produces the following output:

```
S1C88 control program va.b rc        SN00000000-003 (c) year TASKING, Inc.
+ c88 -e -g -Ml -o calc.src calc.c
+ as88 -e -g -o calc.obj calc.src
+ lk88 -e -M calc.obj -lcl -lrt -lfp -ocalc.out -Ocalc
+ lc88 -e -M -ocalc.abs calc.out
```

As you can see, if you use the **-tmp** option, the assembly source files and linker output file will be created in your current directory also.

Of course, you will get the same result if you invoke the tools separately using the same calling scheme as the control program.

As you can see, the control program automatically calls each tool with the correct options and controls. The control program is described in detail in Section 1.4, "Compiler Use".

1.1.4.2 Using the Makefile

The subdirectories in the `examples` directory each contain a `makefile` which can be processed by **mk88**. Also each subdirectory contains a `readme.txt` file with a description of how to build the example.

To build the `calc` demo example follow the steps below. This procedure is outlined as a guide for you to build your own executables for debugging.

1. Make the subdirectory `asm` of the `examples` directory the current working directory.
   This directory contains a makefile for building the `calc` demo example. It uses the default **mk88** rules.
2. Be sure that the directory of the binaries is present in the PATH environment variable.
3. Compile, assemble, link and locate the modules using one call to the program builder **mk88**:

   ```
   mk88
   ```

   This command will build the example using the file `makefile`.

To see which commands are invoked by **mk88** without actually executing them, type:

```
mk88 -n
```

This command produces the following output:

```
S1C88 program builder vx.y rz         SN00000000-003 (c) year TASKING, Inc.
cc88 -g -M -Ml calc.c -o calc.abs
```

The **-g** option in the makefile is used to instruct the C compiler to generate symbolic debug information. This information makes debugging an application written in C much easier to debug.
The **-M** option in the makefile is used to create the linker list file (`.lnl`) and the locator map file (`.map`).
The **-Ml** option specifies to use the large memory model.
The **-o** option specifies the name of the output file.

To remove all generated files type:

```
mk88 clean
```

1.2 Language Implementation

1.2.1 Introduction

The C cross-compiler (**c88**) offers a new approach to high-level language programming for the S1C88 family. It conforms to the ANSI standard, but allows you to control the special functions of the S1C88 in C.
This chapter describes the C language implementation in relation to the S1C88 architecture.

The extensions to the C language in **c88** are:

_sfrbyte and _sfrword
Data types for the declaration of Special Function Registers. The compiler does not allocate memory for an `_sfrbyte` or `_sfrword`.

_at
You can specify a variable to be at an absolute address.

storage types
Apart from a memory category (extern, static, ...) you can specify a storage type in each declaration. This way you obtain a memory model-independent addressing of variables in several address ranges (`_near`, `_far`, `_rom`).

memory-specific pointers
c88 allows you to define pointers which point to a specific target memory. A pointer can point to `_near` or `_far` memory. Each pointer produces efficient code according to its type.

common functions
When a function is declared as a common function (`_common` keyword) then the function contents will be placed within the lower 32K of memory (shared code bank).

assembly functions
See Section 1.2.15, "Calling Assembly Functions", for details.

interrupt functions
You can specify interrupt functions directly through interrupt vectors in the C language (`_interrupt` keyword).

intrinsic functions
A number of pre-declared functions can be used to generate inline assembly code at the location of the intrinsic (built-in) function call. This avoids the overhead which is normally used to do parameter passing and context saving before executing the called function.

1.2.2 Accessing Memory

The S1C88 has a different banking mechanism for CODE and DATA access. The compiler takes care of handling this.

In practice the majority of the C code of a complete application is standard C (without using any language extension). You can compile this part of the application without any modification, using the memory model which fits best to the requirements of the system (code density, amount of external RAM etc.).

Only a small part of the application uses language extensions. These parts often have some of the following properties. They

- access I/O, using the special function registers
- need high execution speed
- need high code density
- access non-default memory
- are used to service interrupts

1.2.2.1 Storage Types

Static storage specifiers can be used to allocate **static** objects in a particular memory area of the addressing space of the processor. All objects taking **static** storage may be declared with an explicit storage specifier. By default static variables will be allocated in `_far` memory for the large and compact code model and in `_near` memory for the small and compact data model.

c88 recognizes the following storage type specifiers:

Storage Type	Description
_near	lowest 64K addresses of data memory
_far	anywhere in data memory, but within one 64K page
_rom	located in ROM

Examples:

```
int _near         Var_in_near;         /* fast accessible integer in low
                                          (64K) addresses of _near Memory */
int _near * _far  Ptr_in_far_to_near;  /* allocate pointer in _far Memory,
                                          used to point to integers in
                                          _near */
char _rom         string[] = "S1C88";  /* string in ROM Memory */
```

Using the `_near` addressing qualifier, allows the compiler to generate faster access code for frequently used variables. Pointers to `_near` memory are also faster in use than pointers to `_far` memory.

Functions are by default allocated in ROM Memory; the storage specifier may be omitted in that case. Also, function return values cannot be assigned to a storage area.

In addition to static storage specifiers, a static object can be assigned to a fixed memory address using the `_at()` keyword:

```
int _near       myvar _at(0x100);
```

This is useful to interface to object programs using fixed memory schemes.

Examples using storage specifiers:

Some examples of using storage specifiers:

```
int _near *p;          // pointer to int in _near memory
                       // (pointer has 16-bit size)
int _far *g;           // pointer to int in _far memory
                       // (pointer has 24-bit size)

g = p;                 /* the compiler issues a warning */
```

If a library function declares:

```
extern int _near foo;  //extern int in _near memory
```

and a data object is declared as:

```
int _far foo;          //int in _far memory
```

the linker will flag this as an error. The usage of the variables is always without a storage specifier:

```
char _far example;     /* define a char in _far memory */
example = 2;           /* assign example               */
```

The generated assembly would be:

```
LD      ep,#@dpag(_example)
LD      a,#2
LD      [@doff(_example)],a
```

All allocations with the same storage specifiers are collected in units called 'sections'. The section with the `_near` attribute will be located within the first 64K. It is always possible to steer the location of sections manually.

Storage and section relations

The following tables show the resulting assembler section types and attributes for each C storage type:

Storage Type	S1C88 Section Type / Attribute
_near	DATA, SHORT
_far	DATA, FIT 10000H
_rom	CODE, ROMDATA, FIT 10000H
const _near	CODE, ROMDATA
const _far	CODE, ROMDATA, FIT 10000H

1.2.2.2 Memory Models

c88 supports four memory models: small, compact code, compact data and large. You can select one of these models with the **-M** option. By default the compiler compiles for the small model. Programs for the S1C88 are always compiled using a reentrant model. Static model functions have to be specified within the source.

The following table gives an overview of the different memory models. If no memory model is specified on the command line, **c88** uses the *small* model because this model generates the most efficient code. The different compiler models assume program/data sizes as follows:

Memory Model	Program Size	Data Size	Description
small (s) (default)	≤ 64K	≤ 64K	No 'LD NB', expand page registers assumed zero
compact code (c)	≤ 64K	> 64K	No 'LD NB', expand page registers are loaded when needed
compact data (d)	> 64K	≤ 64K	'LD NB' are inserted, expand page registers assumed zero
large (l)	> 64K	> 64K	'LD NB' are inserted, expand page registers are loaded when needed

Note that the assembler uses the same model selection assumptions. This ensures that two objects built for different memory models cannot be linked together.

Separate versions of the C and run-time libraries are supplied for all supported models, avoiding the need for you to recompile or rebuild these when using a particular model.

The different models are designed for using the various CPU modes of the S1C88. Because the pushed return addresses differ for the CPU modes, the compiler has to take care of this. The compiler models 'small' and 'compact code' assume two byte return addresses to be pushed on a CALL instruction, while the other models assume three byte return addresses. Please note that the startup module may need to be adapted for your own situation.

The C compiler models are designed for the following CPU modes:

Compiler Model	CPU Mode
small	Single chip (MCU), 64K (MPU)
compact code	512K Min
compact data	512K Max
large	512K Max

Because the PAGE registers handling differs within the different compiler models, and because the return address sizes are different, the S1C88 tools do not accept mixing memory models within a single application. The linker will notice when an application is linked using mixed models.

In all models, C function parameters and automatics are passed via the stack. The linker is using a function call graph of the entire application for this purpose. Data areas of functions which are not calling each other can be overlayed, since these functions will never be active simultaneously. However, this cannot be accomplished for functions called through pointers.

_MODEL

c88 introduces the predefined preprocessor symbol _MODEL. The value of _MODEL represents the memory model selected (**-M** option). This can be very helpful in making conditional C code in one source module, used for different applications in different memory models. See also Section 1.2.21, "Portable C Code", explaining the include file `c88.h`.

The value of _MODEL is:

small model	's'
compact code model	'c'
compact data model	'd'
large model	'l'

Example:

```
#if _MODEL == 's'   /* small model */
...
#endif
```

1.2.2.3 The _at() Attribute

In C for the S1C88 it is possible to place certain variables at absolute addresses. Instead of writing a piece of assembly code, a variable can be placed on an absolute address using the `_at()` attribute.

Example:

```
_far unsigned char Display _at( 0x2000 );
```

The example above creates a variable with the name `Display` at address 0x2000. In the generated assembly code an absolute section will appear. On this position space is reserved for the variable `Display`.

A number of restrictions are in effect when placing variables on an absolute address:

- Only global variables can be placed on absolute addresses. Parameters of functions, or automatics within functions cannot be placed on an absolute address.
- When declared 'extern', the variable is not allocated by the compiler. When the same variable is allocated within another module but on a different address, the compiler, assembler or linker will not notice.
- When the variable is declared 'static', no public symbol will be generated (normal C behavior).
- Absolute variables cannot be initialized, except for absolute variables declared in rom.
- Functions cannot be declared absolute.
- Absolute variables cannot overlap each other, declaring two absolute variables on the same address will cause an error generated by the assembler or by the linker. The compiler does not check this.
- Declaring the same absolute variable within two modules will also produce conflicts during link time (except when one of the modules declares the variable 'extern').

1.2.3 Data Types

The following ANSI C data types are supported. In addition to these types, the `_sfrbyte` and `_sfrword` types are added. Two types of pointers are recognized.

Data Type	Size (in bytes)	Range
signed char	1	-128 to +127
unsigned char	1	0 to 255U
_sfrbyte	1	0 to 255U
signed short	2	-32768 to +32767
unsigned short	2	0 to 65535U
signed int	2	-32768 to +32767
unsigned int	2	0 to 65535U
_sfrword	2	0 to 65535U
signed long	4	-2147483648 to +2147483647
unsigned long	4	0 to 4294967295UL
enum	2	0 to 65535U
_near pointer	2	0 to 65535U
_far pointer	3	0 to 16M

- `char`, `_sfrbyte`, `_sfwordvshort`, `int` and `long` are all integral types, supporting all implicit (automatic) conversions.
- **c88** generates instructions using (8 bit) character arithmetic, when it is correct to evaluate a character expression this way. This results in a higher code density compared with integer arithmetic. Section 1.2.3.2, "Character Arithmetic", provides details.
- `char` and `short` are treated as 8-bit and 16-bit `int` respectively.
- the S1C88 convention is used, storing variables with the most significant part at the higher memory address (Little Endian).

1.2.3.1 ANSI C Type Conversions

According to the ANSI C X3.159-1989 standard, a character, a short integer, an integer bit field (either signed or unsigned), or an object of enumeration type, may be used in an expression wherever an integer may be used. If a `signed int` can represent all the values of the original type, then the value is converted to `signed int`; otherwise the value will be converted to `unsigned int`. This process is called *integral promotion*.

Integral promotion is also performed on function pointers and function parameters of integral types using the old-style declaration. To avoid problems with implicit type conversions, you are advised to use function prototypes.

Many operators cause conversions and yield result types in a similar way. The effect is to bring operands into a common type, which is also the type of the result. This pattern is called the *usual arithmetic conversions*.

Integral promotions are performed on both operands; then, if either operand is `unsigned long`, the other is converted to `unsigned long`.
Otherwise, if one operand is `long` and the other is `unsigned int`, the effect depends on whether a `long` can represent all values of an `unsigned int`; if so, the `unsigned int` operand is converted to `long`; if not, both are converted to `unsigned long`.
Otherwise, if one operand is `long`, the other is converted to `long`.
Otherwise, if either operand is `unsigned int`, the other is converted to `unsigned int`.
Otherwise, both operands have type `int`.

See also Section 1.2.3.2, "Character Arithmetic".

Note that sometimes surprising results may occur, for example when unsigned char is promoted to int. You can always use explicit casting to obtain the type required. The following example makes this clear:

```
static unsigned char a=0xFF, b, c;

void f()
{
        b=~a;
        if ( b == ~a )
        {
                /* This code is never reached because,
                 * 0x0000 is compared to 0xFF00.
                 * The compiler converts character 'a' to
                 * an int before applying the ~ operator
                 */
                ...
        }

        c=a+1;
        while( c != a+1 )
        {
                /* This loop never stops because,
                 * 0x0000 is compared to 0x0100.
                 * The compiler evaluates 'a+1' as an
                 * integer expression. As a side effect,
                 * the comparison will also be an integer
                 * operation
                 */
                ...
        }
}
```

To overcome this 'unwanted' behavior use an explicit cast:

```
static unsigned char a=0xFF, b, c;

void f()
{
        b=~a;
        if ( b == (unsigned char)~a )
        {
                /* This code is always reached */
                ...
        }

        c=a+1;
        while( c != (unsigned char)(a+1) )
        {
                /* This code is never reached */
                ...
        }
}
```

Keep in mind that the arithmetic conversions apply to multiplications also:

```
static int      h, i, j;
static long     k, l, m;

/* In C the following rules apply:
 *              int  * int      result: int
 *              long * long     result: long
 *
 *      and NOT int  * int      result: long
 */

void f()
{
        h = i * j;              /* int  * int  = int    */
        k = l * m;              /* long * long = long   */

        l = i * j;              /* int * int = int,
                                 * afterwards promoted (sign
                                 * or zero extended) to long
                                 */
        l = (long) i * j;       /* long * long = long   */
        l = (long)(i * j);      /* int * int = int,
                                 * afterwards casted to long
                                 */
}
```

1.2.3.2 Character Arithmetic

c88 generates code using 8 bit (character) arithmetic as long as the result of the expression is exactly the same as if it was evaluated in integer arithmetic. This must be done, because ANSI does not define character arithmetic and character constants. Although the S1C88 performs 16-bit operation as fast as 8-bit operations, the overhead caused by the integral promotions is suppressed.
So it is recommended to use character variables in expressions, because it saves data space for allocation, and often results in a higher code density. You can always force to use character arithmetic with a character cast. The following examples clarify when integer arithmetic is used and when character arithmetic:

```
char    a, b, c, d;
int     i;

void
main()
{
        c = a + b;                      /* character arithmetic */
        i = a + b;                      /* integer arithmetic   */
        i = (char)(a + b);              /* character arithmetic */

        c = a / d;                      /* character arithmetic */
        c = (a + b) / d;                /* integer arithmetic   */
        c = ((char)(a + b)) / d;        /* character arithmetic */

        c = a >> d;                     /* character arithmetic */
        c = (a + b) >> d;               /* integer arithmetic   */

        if ( a > b )                    /* character arithmetic */
                c = d;
        if ( (a + b) > c )              /* integer arithmetic   */
                c = d;
}
```

1.2.3.3 Special Function Registers

The `_sfrbyte` and `_sfrword` keywords allow direct access to all special function registers, as if they were C variables. These special function registers can be used the same way as any other integral data type, including all automatic conversions.

An `_sfrbyte` is handled the same way as a `volatile unsigned char` variable. An `_sfrword` is handled as a `volatile unsigned int` variable.

You can also declare sfr-registers within your C source by using the data types `_sfrbyte` or `_sfrword`. The notation is as follows:

```
_sfrbyte        name _at( address )    ;
_sfrword        name _at( address )    ;
```

where, *name* must be replaced with the name of the sfr-register you want to specify. *address* is the byte or word address of the sfr-register. Because these registers are placed in the sfr-area of the processor, the compiler will not allocate any storage space.

Note, that the words 'sfrbyte' and 'sfrword' are not reserved words for **c88**. So these words can be used as identifiers. **c88** does not generate symbolic debugging information for special function registers, because they are already known by the debugger.

Because the special function registers are dealing with I/O, it is not correct to optimize away the access to them. Therefore, **c88** deals with the special function registers as if they were declared with the `volatile` qualifier.

For example:

```
_sfrbyte        SPP      _at(0xFF01);

int             i;
volatile int    v;

main()
{
        i;              /* optimized away                 */
        SPP=1;          /* access SPP (implicit volatile) */
        v;              /* volatile: access variable      */
}
```

1.2.4 Function Parameters

c88 supports (ANSI) prototyping of function parameters. Therefore, **c88** allows passing parameters of type `char`, **without** converting these parameters to `int` type. This results into higher code density, higher execution speed and less RAM data space needed for parameter passing.

For example, in the following C code:

```
void    func( char number, long value );
int     printf( char *format, ... );

void
main(void)
{
        int     i;
        char    c;

        func( c, i );
        printf( "c=%d, i=%d\n", c, i );
}
```

the code generator uses the prototype of `func()` and:

- passes `c` as a byte
- promotes `i` to long before passing it as a long

However, the code generator does not know anything of the `printf()` arguments, because this function is declared with a variable argument list. If there is no prototype (as with the old style K & R functions), the compiler promotes both char type parameters to int type, the same way an automatic conversion is done in an assignment of a char type variable to an int type variable. So, with the `printf()` call the code generator:

- promotes `c` to int before passing it as int
- passes `i` as int

1.2.5 Parameter Passing

By default parameters are passed via registers. If not enough registers are available, some parameters are passed via registers, the other parameters are passed over the stack.

All parameters of a variable argument list function are always passed over the stack. Parameters are pushed in reverse order, so all ANSI C macros defined in `stdarg.h` can be applied.

Example with variable argument function:

```
_printf( char *format, ... )
```

- all parameters (including `format`) are passed via the stack.

1.2.6 Automatic Variables

In non-reentrant functions recursion is not possible. In these functions automatic variables are not allocated on a stack, but in a static area. In a reentrant function automatic variables are treated the conventional way: coming and going with a function on the stack. In static functions it is possible to force an automatic to a specified memory by using a storage type specifier. The automatics are still overlayable with automatics of other functions.

Although automatic variables are allocated in a static area with non-reentrant functions, they are **not** the same as local variables (within a function) which are declared to be static by means of the `static` keyword.

The difference is:

- (as in the 'normal' approach) it is not guaranteed that an automatic variable still has the same value as the previous time the function returned, because it may have been overlaid with another automatic variable of another module.
- (as in the 'normal' approach) it is guaranteed that the value of the static variable is the same as the previous time the function returned. Static variables are never overlaid.

1.2.7 Register Variables

In C the `register` type qualifier tells the compiler that the variable will be used very often. So the code generator must try to reserve a register for this variable and use this register instead of the stack location of this automatic variable. Whenever possible, the compiler allocates automatic objects and parameter objects within registers. **c88** therefore ignores the `register` keyword.

For every object not placed in registers, the next rules apply.

Reentrant functions:

For these functions, automatic variables are allocated on the stack and are addressed using the stacked or indexed addressing mode.

The code generator of **c88** uses a 'saved by caller' strategy. This means that a function which needs the contents of one or more registers over a function call, must save the contents of these 'registers' and restore them after the call. The major advantage of this approach is, that only registers which are really used after the call are saved.

Conclusion:

The usage of the `register` keyword is not necessary for improving code density or speed.

Note: The register type qualifier cannot be used for arrays and structures.

1.2.8 Initialized Variables

Non automatic initialized variables use the same amount of space in both ROM and RAM (for all possible RAM memory spaces). This is because the initializers are stored in ROM and copied to RAM at start-up. This is completely transparent to the user. The only exception is an initialized variable residing in ROM, by means of the `_rom` storage type specifier:

Examples (small memory model)**:**

```
int        i = 100;            /* 2 bytes in ROM and
                                  2 bytes in RAM           */
_rom  int  j = 3;              /* 2 bytes in ROM           */
char      *p = "TEXT";         /* 2 bytes for p in RAM
                                  5 bytes for "TEXT" in ROM */
_rom  char a[] = "HELP";       /* 5 bytes in ROM           */
_near char c = 'a';            /* 1 byte in ROM
                                  1 byte in _near RAM      */
```

1.2.9 Type Qualifier volatile

You can use the `volatile` type qualifier when modifications on the object have undesired side effects when they are performed in the regular way. Memory locations may not be updated because of compiler optimizations, which attempt to save a memory write by keeping the value in a register. When a variable is declared with the `volatile` qualifier, the compiler disables such optimizations. Volatile variables are located in a segment of which the NOCLEAR attribute is set.

The ANSI report describes that the updates of volatile objects follow the rules of the abstract machine (the target processor) and thus access to a volatile object becomes implementation defined.

Example:

```
const volatile _near int real_time_clock _at(0x1234);

/*      define the real time clock register;
        it is read-only (const);
        read operations must access the real memory
        location (volatile)
*/
```

1.2.10 Strings

In this section the word 'strings' means the separate occurrence of a string in a C program. So, array variables initialized with strings are just initialized character arrays, which can be allocated in any memory type, and are not considered as 'strings'. See Section 1.2.8, "Initialized Variables", for more information on this topic.

Strings and literals in a C source program, which are not used to initialize an array, have static storage duration. The ANSI X3.159-1989 standard permits string literals to be put in ROM. **c88** always allocates a string in ROM. Note that initialized arrays are still located in RAM.

```
char ramhelp[] = "help";
/* allocation of 5 bytes in RAM and 5 bytes in ROM */
```

Example of an array in ROM only, initialized with the addresses of strings, also ROM only:

```
char * _rom message[] = {"hello","alarm","exit"};
```

ANSI string concatenation is supported: adjacent strings are concatenated - only when they appear as primary expressions - to a single new one. The result may not be longer than the maximum string length (ANSI limit 509 characters, actual compiler limit 1500 characters).

The ANSI Standard states that identical string literals need not be distinct, i.e. may share the same memory. Because memory can be very scarce with microcontroller applications, the **c88** compiler overlays identical strings within the same module.

In section 3.1.4 the Standard states that behavior is undefined if a program attempts to modify a string literal. Because it is a common extension to ANSI (A.6.5.5) that string literals are modifiable, there may be existing C source modifying strings at run-time. This can be done either with pointers, or even worse:

```
"st ing"[2] = 'r';
```

c88 does not accept this statement.

1.2.11 Pointers

Some objects have two types: a 'logical' type and a storage type. For example, a function is residing in ROM (storage type), but the logical type is the return type of this function. The most obvious C type having different storage and logical type is a pointer. For example:

```
_far char *_near p;    /* pointer residing in _near, pointing to _far */
```

means `p` has storage type `_near` (allocated in direct addressable RAM), but has logical type 'character in target memory space `_far`'. The memory type specifier used left to the '*', specifies the target memory of the pointer, the memory specifier used right to the '*', specifies the storage memory of the pointer.

The memory type specifiers are treated like any other type specifier (like unsigned). This means the pointer above can also be declared (exactly the same) using:

```
char _far *_near p;    /* pointer residing in _near, pointing to _far */
```

If the target memory and storage memory of a pointer are not explicitly declared, **c88** uses the default of the memory model selected.

Model	Target Memory Default
's'	_near
'c'	_far
'd'	_near
'l'	_far

In pointer arithmetic **c88** checks, besides the type of each pointer, also the target memory of the pointers, which should be the same. For example, it is invalid (and has no use) to assign a pointer to `_far` to a pointer to `_near`. Of course, an appropriate cast corrects the error.

1.2.12 Function Pointers

Reentrant functions use the stack for passing parameters and automatic variable allocation. When using the reentrant memory model, all functions are in fact implicitly reentrant.

So, function pointers are only allowed to point to functions compiled as reentrant. Parameters are passed to these functions via the stack. A function pointer may point to any reentrant function in the application.

1.2.13 Inline C Functions

The `_inline` keyword is used to signal the compiler to inline the function body instead of calling the function. An inline function must be defined in the same source file before it is 'called'. When an inline function has to be called in several source files, each file must include the definition of the inline function. Usually this is done by defining the inline function in a header file.

Not using a function which is defined as an `_inline` function does not produce any code.

Example (`t.c`)**:**

```
int     w,x,y,z;

_inline int
add( int a, int b )
{
        return( a + b );
}

void
main( void )
{
        w = add( 1, 2 );
        z = add( x, y );
}
```

No specific debug information is generated about inline functions. The debugger cannot step-into an inline function, it considers the inline function as one HLL source line.

The pragmas `asm` and `endasm` are allowed in inline functions. This makes it possible to define inline assembly functions. See also Section 1.2.14, "Inline Assembly", in this chapter.

1.2.14 Inline Assembly

c88 supports inline assembly using the following pragmas:

#pragma asm	Insert assembly text following this pragma.
#pragma asm_noflush	As #pragma asm, but the peephole optimizer does not flush the code buffer.
#pragma endasm	Switch back to the C language.

Note that C modules containing inline assembly are not portable and are very hard to prototype in other environments.

The peephole optimizer in the compiler maintains a code buffer for optimizing sequences of assembly instructions before they are written in the output file. The compiler does not interpret the text of inline assembly. It passes inline assembly lines directly to the output file. To prevent that instructions in the peephole buffer, which belong to C code before the inline assembly lines, will be written in the output file after the inline assembly text, the compiler flushes the instruction buffer in the peephole optimizer. All instructions in the buffer are written to the output file. If this behavior is not desired the pragma **asm_noflush** starts inline assembly without flushing the code buffer.

1.2.15 Calling Assembly Functions

The S1C88 C compiler uses fixed registers for passing arguments to functions (see Section 1.3.2, "Register Usage"). When calling assembler functions from a C program, follow this scheme. However, all the arguments must be passed via the usable registers only. (Note that an error or warning will not occur even if some arguments cannot be allocated to the registers.)

When the C functions that call assembler functions are compiled, make sure the register allocation of the parameters that are passed to the assembler functions using the compile results (assembler source). The following is a program example.

1. **Source program**

```
int sub_asm(int ia, char ca, int ib, char cb, int _near *pic);

int main(void)
{
        int ia, ib, ic, id;
        int _near *pic;
        char ca, cb;

        ia = 1;
        ib = 2;
        ca = '3';
        cb = '4';
        ic = 5;
        pic = &ic;

        id = sub_asm(ia, ca, ib, cb, pic);

        id +=1;
              :
}

#pragma asm
_sub_asm:
              :
ret
#pragma endasm
```

2. **Assembler source after compiled (when no option is specified)**

```
        LD      iy,#05h
        LD      [sp],iy                 ; [sp] <- ic
        LD      iy,sp                   ; iy <- &ic
        LD      ba,#01h                 ; ba <- ia
        LD      l,#033h                 ; l <- ca
        LD      ix,#02h                 ; ix <- ib
        LD      yp,#034h                ; yp <- cb
        CARL    _sub_asm
        INC     ba                      ; id <- ba
              :
_sub_asm:
              :
ret
```

3. **Parameter allocation scheme** (see also Section 1.3.2, "Register Usage")

   Arguments are passed via the registers shown below in the order of descending priorities.

	Priority Higher	Lower
char	A L YP XP H B	
int	BA HL IX IY	
long	HLBA IYIX	
near pointer	IY IX HL BA	
far pointer	IYP IXP HLP	

Consequently, the function in the sample above,

```
int sub_asm(int ia, char ca, int ib, char cb, int *pic);
```

uses the registers listed below.

```
BA = int  ia
L  = char ca
IX = int  ib
YP = char cb
IY = int  *pic
```

(Since the A register that has the highest priority for char type arguments is used for `int ia`, `char ca` is allocated to the L register that has second priority. The same is applied to other arguments.)

The int type return value from `sub_asm` will be allocated to the BA register that has the highest priority for int type values.

1.2.16 Intrinsic Functions

When you want to use some specific S1C88 instructions, that have no equivalence in C, you would be forced to write assembly routines to perform these tasks. However, **c88** offers a way of handling this in C. **c88** has a number of built-in functions, which are implemented as intrinsic functions.

To the programmer intrinsic functions appear as normal C functions, but the difference is that they are interpreted by the code generator, so that more efficient code may be generated. Several pre-declared functions are available to generate inline assembly code at the location of the intrinsic function call. This avoids the overhead that is normally introduced by parameter passing and context saving before executing the called function.

The names of the intrinsic functions all have a leading underscore, because the ANSI specification states that public C names starting with an underscore are implementation defined.

The advantages of using intrinsic functions, compared with in-line assembly (pragma asm/endasm) are:

- the possibility to use simulation routines or stub functions by a host compiler, to replace the inline assembly code generated by **c88**
- C level variables can be accessed
- the compiler chooses to generate the most efficient code to access C variables
- intrinsic code is optimized, except for `_nop()`

The following intrinsic functions are implemented:

Function	Description
_bcd()	Set 'D' flag on expression evaluation
_halt()	HALT instruction
_int()	Software interrupt
_jrsf()	Jump to relative location if condition is true
_nop()	NOP instruction, not optimized away
_pack()	Pack integer into character value
_rlc()	Rotate left
_rrc()	Rotate right
_slp()	SLP instruction
_swap()	Swap high and low nibbles
_ubcd()	Set 'U' and 'D' flags on expression evaluation
_unpack()	Set 'U' flag on expression evaluation
_upck()	Unpack character into integer value

Prototypes for the intrinsic functions are present in `c88.h`. Below are the details of the implemented intrinsic functions:

_bcd

`void _bcd();`

When evaluating the argument expression, the 'D' flag will be set. That means, Add/Subtract and Negate instructions are done as binary decimal. Problems can be expected when the expression uses for example floating point while the 'D' flag will not be reset throughout the expression evaluation.

The argument may be of any type of expression (`char/int/long`). Therefore, the argument list is implemented as an old-style (K&R style) function definition (without defining argument type).

Returns nothing.

_halt

`void _halt( void );`

Generate HALT instructions.

Returns nothing.

```
_halt();
```

... Code ...

```
        HALT
```

_int

`void _int( ICE vector );`

Insert an execute software interrupt instruction (INT). The argument should be an ICE type value, determining the interrupt vector address to jump to. ICE denotes that the operand must be an Integral Constant Expression rather than any type of integral expression.

Returns nothing.

_jrsf

`char _jrsf( ICE number );`

Use the JRS F*number*,_lab instruction. This instruction is ideal for use within an if() condition test. The given number must be a constant value between 0 and 3. ICE denotes that the operand must be an Integral Constant Expression rather than any type of integral expression. The code generator chooses between the F*number* and the NF*number* variant of the instruction.

Returns the result.

```
if ( _jrsf( 2 ) )
{
        ...
}
```

... Code ...

```
        JRS NF2, _L0001
_L0001:
```

_nop

`void _nop( void );`

Generate NOP instructions.

Returns nothing.

```
_nop();
```

... Code ...

```
        NOP
```

_pack

char _pack(int *operand*);

Use the PACK instruction to pack the integer *operand* into a character value.

Returns the character value.

_rlc

char _rlc(char *operand*);

Use the RLC instruction to rotate byte *operand* to the left. The instruction only affects the result, not the *operand*.

Returns the result.

```
char c;
int i;
/* rotate left */
c = _rlc( c );
```

_rrc

char _rrc(char *operand*);

Use the RRC instruction to rotate byte *operand* to the right. The instruction only affects the result, not the *operand*.

Returns the result.

```
char c;
int i;
/* rotate right */
c = _rrc( c );
```

_slp

void _slp(void);

Generate SLP instruction.

Returns nothing.

```
_slp();
```

... Code ...

```
      SLP
```

_swap

char _swap(char *operand*);

Use the SWAP instruction to swap the high and low nibbles of the character *operand*.

Returns the result.

_ubcd

void _ubcd();

When evaluating the argument expression, the 'U' and 'D' flags will be set. That means, only the lower nibble of a byte is used to do the computation, and Add/Subtract and Negate instructions are done as a BCD operation. Problems can be expected when the expression uses for example floating point while the 'D' flag will not be reset throughout the expression evaluation.

The argument may be of any type of expression (`char/int/long`). Therefore, the argument list is implemented as an old-style function definition.

Returns nothing.

_unpack

`void _unpack();`

When evaluating the argument expression, the 'U' flag will be set. That means, only the lower nibble of a byte is used to do the computation.

The argument may be of any type of expression (`char/int/long`). Therefore, the argument list is implemented as an old-style function definition.

Returns nothing.

_upck

`int _upck( char operand );`

Use the UPCK instruction to unpack the character *operand* into an integer value.

Returns the integer value.

1.2.17 Interrupts

The S1C88 C language introduces a new reserved word: `_interrupt`, which can be seen as a special type qualifier, only allowed with function declarations. A function can be declared to serve as an interrupt service routine. Interrupt functions cannot return anything and must have a **void** argument type list. For example, in:

```
void _interrupt(vector)
_isr(void)
{
        ...
};
```

The compiler generates an interrupt service frame for interrupts. The `_interrupt` function qualifier takes one argument, *vector*, that defines the interrupt vector address of a two byte interrupt vector area.

Some interrupts are reserved and handled or used by the compiler (run-time library) like:

- Hardware reset.

Example of _interrupt:

Suppose, you want an interrupt function for a software interrupt, and the vector address is 0x30:

```
int c;

void
_interrupt( 0x30 )
transmit(void)
{
        c = 1;
}
```

This will result in assembly:

```
        DEFSECT ".abs_48", CODE AT 48
        SECT    ".abs_48"
        DW      _transmit

        DEFSECT ".short_code", CODE, SHORT
        SECT    ".short_code"
_transmit:
        LD      A,#1
        LD      [_c],A
        RETE
```

1.2.18 Structure Tags

A tag declaration is intended to specify the lay-out of a structure or union. If a memory type is specified, it is considered to be part of the declarator. A tag name itself, nor its members can be bound to any storage area, although members having type "... pointer to" do require one. A tag may then be used to declare objects of that type, and may allocate them in different memories (if that declaration is in the same scope). The following example illustrates this constraint.

```
struct S {
       _near int i;           /* referring to storage: not correct */
       _far char *p;          /* used to specify target memory: correct */
       };
```

In the example above **c88** ignores the erroneous `_near` storage specifier (without displaying a warning message).

1.2.19 Typedef

Typedef declarations follow the same scope rules as any declared object. Typedef names may be (re-) declared in inner blocks but not at the parameter level. However, in typedef declarations, memory specifiers are allowed. A typedef declaration should at least contain one type specifier.

Examples:

```
typedef _near int NEARINT; /* storage type _near: OK */
typedef int _far *PTR;     /* logical type _far storage type 'default' */
```

1.2.20 Language Extensions

The following language extensions are implemented in the S1C88 C Compiler. They cannot be translate with any ANSI-C conforming C-compiler.

Character arithmetic

Perform character arithmetic. **c88** generates code using 8-bit character arithmetic as long as the result of the expression is exactly the same as if it was evaluated using integer arithmetic. See also Section 1.2.3.2, "Character Arithmetic".

Uninitialized constant definitions

Define storage for uninitialized constant rom data, instead of implicit zero initialization. The compiler generates a '`DS 1`' for '`const char i[1];`'.

Keyword language extensions

Allow keyword language extensions such as `_near`, `_far` and `_sfrbyte`.

Maximum number of significant characters

500 significant characters are allowed in an identifier instead of the minimum ANSI-C translation limit of 31 significant characters. Note: more significant characters are truncated without any notice.

C++ style comments

Allow C++ style comments in C source code. For example:

```
// e.g this is a C++ comment line.
```

__STDC__ definition

__STDC__ is defined as '0'. The decimal constant '0', intended to indicate a non-conforming implementation.

Promoting old-style function parameters

Do not promote old-style function parameters when prototype checking.

Using unsigned char

Use type unsigned char for 0x80–0xff. The type of an unsuffixed octal or hexadecimal constant is the first of the corresponding list in which its value can be represented:

```
char, unsigned char, int, unsigned int, long, unsigned long
```

lvalue cast

Allow type cast of an lvalue object with incomplete type `void` and lvalue cast which does not change the type and memory of an lvalue object.

Example:

```
void *p; ((int*)p)++;  /* allowed */
int  i;  (char)i=2;    /* NOT allowed */
```

Checking assignments of a constant string to a non-constant string pointer

Do not check for assignments of a constant string to a non-constant string pointer. With this option the following example produces no warning:

```
char *p;
void main( void ) { p = "hello"; }
```

1.2.21 Portable C Code

If you are developing C code for the S1C88 using **c88**, you might want to test some code on the host you are working on, using a C compiler for that host. Therefore, we deliver the include file `c88.h`. This header file checks if _C88 is defined (**c88** only), and redefines the storage type specifiers if it is not defined.

When using this include file, you are able to use the storage type specifiers (when needed) and yet write 'portable C code'.

Furthermore an adapted prototype of each S1C88 C built-in function is present, because these functions are not known by another ANSI compiler. If you use these functions, you should write them in C, performing the same job as the S1C88 and link these functions with your application for simulation purposes.

1.2.22 How to Program Smart

If you want to get the best code out of **c88**, the following guidelines should be kept in mind:

1. Always use function prototyping. So, `char` variables can be passed as `char` without being promoted to `int`.
2. If you are using the large model (because it is not possible to use a smaller model), try to declare the most frequently used variables (static) with storage type `_near`. If you want your code to remain portable, you can use the `register` keyword.
3. Try to use the `unsigned` qualifier as much as possible (e.g. `for (i = 0; i < 500; i++)`), because unsigned comparisons require less code than signed comparisons.
4. Try to use the smallest data type as possible: character for small loops and so on. See also Section 1.2.3.2, "Character Arithmetic".

1.3 Run-time Environment

1.3.1 Startup Code

When linking your C modules with the library, you automatically link the object module, containing the C startup code. This module is called `cstart.obj` and is present in every C library (once for every model and execution mode).

Because this module specifies the run-time environment of your S1C88 C application, you might want to edit it to match your needs. Therefore, this module is delivered in source in the file `cstart.c` in the `src` subdirectory of the `lib` directory. Typically, you will copy the template startup file to your own directory and edit it. The startup code contains macro preprocessor symbols to tune the startup code. The invocation (using the **cc88** control program) is:

```
cc88 -Ms -c cstart.c
```

In the C startup code an absolute code section is defined for setting up the reset vector and the S1C88 C environment. The reset vector contains a jump to the _START label. This global label may not be removed, since it is referred to by the C compiler. It is also used as the default start address of the application (see the `start` keyword in the locator description language DELFEE). The code space for all non used interrupt vectors are reserved in the locator description file to prevent **lc88** from using this area for a user code section. This code space may be used for small user code sections.

The stack is defined in the locator description file (`.dsc` in directory `etc`) with the keyword `stack`, which results in a section called `stack`. See Section 1.3.4, "Stack", for detailed information on the stack.

The heap is defined in the description file with the keyword `heap`, which results in as section called `heap`. See Section 1.3.5, "Heap", for detailed information on heap management.

The startup code also takes care of initialized C variables, residing in the different RAM areas. Each memory type has a unique name for both the ROM and the RAM section. The startup code copies the initial values of initialized C variables from ROM to RAM, using these special sections and some run-time library functions. When initialization of C variables is not needed, you can translate the file `cstart.c` with **-DNOCOPY**. See also the `table` keyword in the locator description language DELFEE.

When everything described above has been executed, your C application is called, using the global label `_main`, which has been generated by **c88** for the C function `main()`.

When the C application 'returns', which is not likely to happen in an embedded environment, the program ends with a SLP instruction, using the assembly label `__exit`. When using a debugger, it can be useful to set a breakpoint on this label, indicating the program has reached the end, or the library function `exit()` has been called.

One extra feature is done in the startup code for the S1C88 microcontroller. The Watchdog timer is handled. A very common problem is that the NMI/Watchdog interrupt handling is forgotten in an application. This causes unexpected results, as the Watchdog cannot be disabled. Therefore, this handling is by default done by the startup code. When an application needs to handle the NMI/Watchdog itself, then the startup code needs to be recompiled for that application.

The following macro can be used to control the functionality of `cstart.c`:

NOCOPY - Do NOT produce code to clear BSS sections and initialize DATA sections.

1.3.2 Register Usage

c88 will try to use the available registers as efficient as possible. The compiler uses a flexible register allocation scheme, which implies that any change to the C code may result in a different register usage.

For passing parameters to functions **c88** uses a fixed scheme:

- The arguments are passed via the registers A, B, L, H, YP, XP, BA, HL, IX and IY. Char arguments are passed via the byte registers A, L, YP, XP, H and B. Integers are passed via the word registers BA, HL, IX and IY. Long arguments are passed via 32-bit register pairs HLBA and IYIX.
- Structures and unions are passed via the stack.
- Near pointers are passed in registers IY, IX HL and BA. Far pointers are passed in register pairs IYP, IXP and HLP (where, IYP = IY + YP, IXP = IX + XP, HLP = HL + A).
- When there are too much arguments to be passed in the registers the arguments will be passed via the stack.

For C function return types, the following registers are used:

Return Type	Register	Description
char	A	accumulator
short/int	BA	
long	HLBA	(HL high word, BA low word)
pointer	HLP	HL +A

- Structures and unions are returned on the stack.

1.3.3 Section Usage

c88 uses a number of sections. For each used section the compiler generates a DEFSECT directive in the output. The following list gives an overview of section-names used:

Section Name	Comment
.text	model s and c: code
.text_function	model d and l: code
.comm	code with _common qualifier _interrupt code
.nbss	cleared _near data
.fbss	cleared _far data
.nbssnc	non-cleared _near data
.fbssnc	non-cleared _far data
.ndata	initialized _near data
.fdata	initialized _far data
.nrdata	const _near data
.frdata	const _far data

1.3.4 Stack

The S1C88 processor has a **system stack** of a maximum of 64K byte. This system stack is used for function calls, interrupts, function parameters and automatics. Static functions use overlayable sections for these purposes.

The following diagram show the structure of the system stack when using reentrant (= default) functions.

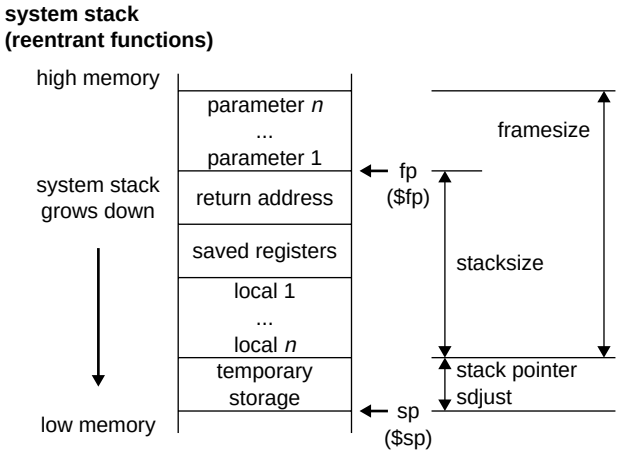


Fig. 1.3.4.1 Stack diagram

The stack is defined in the locator description file (`.dsc` in directory `etc`) with the keyword `stack`, which results in a section called `stack`. The description file tells **lc88** where to allocate the stack.

The stack size can be controlled with the keyword `length=`*`size`* in the description file. If you do not specify the stack size, the locator will allocate the rest of the available RAM for the stack, as done in the startup code. You can use the locator defined labels `__lc_bs` and `__lc_es` in your application to retrieve the begin and end address of the stack. Please note that the locator will only allocate a stack section if the application refers to one of the locator defined symbols `__lc_bs` or `__lc_es`. Remember that there must be enough space allocated for the stack, which grows downwards.

For non-reentrant functions, (non-register) automatics and (non-register) parameters are allocated in a **static area**, and therefore, do not use any stack space.

1.3.5 Heap

The heap is only needed when dynamic memory management library functions are used: `malloc()`, `calloc()`, `free()` and `realloc()`. The heap is a reserved area in memory. Only if you use one of the memory allocation functions listed above, the locator automatically allocates a heap, as specified in the locator description file with the keyword `heap`.

A special section called `heap` is used for the allocation of the heap area. You can place the heap section anywhere in memory, using the locator description file. You can specify the size of the heap using the keyword `length=`*`size`* in the locator description file. If you do not specify the heap size and yet refer to it (e.g. call `malloc()`), the locator will allocate the rest of the available memory for the heap. The locator defined labels `__lc_bh` and `__lc_eh` (begin and end of heap) are used by the library function `sbrk()`, which is called by `malloc()` when memory is needed from the heap.

Example part of the locator description file defining the heap size and location:

```
amode data
{
        section selection=w;
        heap length=1000;  // heap (only when used)
}
```

Note that the special `heap` segment is only allocated when its locator labels are used in the program.

When the heap is needed for an application built in the small or compact data model of the compiler, the locator description file needs to be changed. If not, the locator will report errors. The declaration of the heap should be moved from 'amode data' into 'amode data_short'.

1.3.6 Interrupt Functions

Interrupt functions may be implemented directly in C, by using the `_interrupt(n)` function qualifier. A function declared with this qualifier differs from a normal function definition in a number of ways:

1. All registers that might possibly be corrupted during the execution of the interrupt function are saved on function entry and restored on function exit. Normally, only the registers directly used by the interrupt function will be saved.
2. The function is terminated with a RETE instruction instead of a RET instruction.

Example:

```
; S1C88 C compiler v99.9 r9     SN00000000-000 (c) year TASKING, Inc.
; options: -n -s
$CASE ON

        NAME    INTERPT
;        interpt.c:
; 1     |int flag;
; 2     |
; 3     |_interrupt( 0x30 )
; 4     |void handler( void )
; 5     |{

        GLOBAL  _handler
        DEFSECT ".code48", CODE AT 030H

        SECT    ".code48"
        DW      _handler

        DEFSECT ".comm", CODE, SHORT
        SECT    ".comm"

_handler:
        PUSH    ale
; 6     |    flag=1;

        LD      iy,#01h
        LD      [_flag],iy

; 7     |}

        POP     ale
        RETE

        DEFSECT ".bss", DATA, SHORT, CLEAR
        SECT    ".bss"
        GLOBAL  _flag
_flag:  DS      2
        EXTERN  (DATA) __lc_es
        END
```

1.4 Compiler Use

1.4.1 Control Program

The control program **cc88** facilitates the invocation of the various components of the S1C88 tool chain, from a single command line. The control program accepts source files and options on the command line in random order.

The invocation syntax of the control program is:

cc88 [[*option*] ... [*control*] ... [*file*] ...] ...

Options are preceded by a '-' (minus sign). The input *file* can have one of the extensions explained below.

The control program recognizes the following argument types:

- Arguments starting with a '-' character are options. Some options are interpreted by the control program itself; the remaining options are passed to those programs in the tool chain that accept the option.
- Arguments with a `.c` suffix are interpreted as C source programs and are passed to the compiler.
- Arguments with a `.asm` suffix are interpreted as assembly source files which have to be preprocessed and passed to the assembler.
- Arguments with a `.src` suffix are interpreted as compiled assembly source files. They are directly passed to the assembler.
- Arguments with a `.a` suffix are interpreted as library files and are passed to the linker.
- Arguments with a `.obj` suffix are interpreted as object files and are passed to the linker.
- Arguments with a `.out` suffix are interpreted as linked object files and are passed to the locator. The locator accepts only one `.out` file in the invocation.
- Arguments with a `.dsc` suffix are treated as locator command files. If there is a file with extension `.dsc` on the command line, the control program assumes a locate phase has to be added. If there is no file with extension `.dsc`, the control program stops after linking (unless it has been directed to stop in an earlier phase).
- Everything else is considered an object file and is passed to the linker.

Normally, a control program tries to compile and assemble all source files to object files, followed by a link and locate phase which produces an absolute output file. There are however, options to suppress the assembler, linker or locator stage. The control program produces unique filenames for intermediate steps in the compilation process, which are removed afterwards. If the compiler and assembler are called subsequently, the control program prevents preprocessing of the compiler generated assembly file. Normally, assembly input files are preprocessed first.

The following options are interpreted by the control program:

Control Program Options

Option	Description
-Mc	Compact code memory model
-Md	Compact data memory model
-Ml	Large memory model
-Ms	Small memory model
-Ta *arg*	Pass argument directly to the assembler
-Tc *arg*	Pass argument directly to the C compiler
-Tlk *arg*	Pass argument directly to the linker
-Tlc *arg*	Pass argument directly to the locator
-V	Display version header only
-al	Generate absolute list file
-c	Do not link: stop at `.obj`
-cl	Do not locate: stop at `.out`
-cs	Do not assemble: compile C files to `.src` and stop
-f *file*	Read arguments from *file* ("-" denotes standard input)
-ieee	Set locator output file format to IEEE-695 (default)
-nolib	Do not link with the standard libraries
-o *file*	Specify the output file
-srec	Set locator output file format to Motorola S-records
-tmp	Keep intermediate files
-v	Show command invocations
-v0	Show command invocations, but do not start them

1.4.1.1 Detailed Description of the Control Program Options

-M{s | c | d | l} Specify the memory model to be used:

small	(**s**)
compact data	(**d**)
compact code	(**c**)
large	(**l**)

-Ta *arg*
-Tc *arg*
-Tlk *arg*
-Tlc *arg*

With these options you can pass a command line argument directly to the assembler (**-Ta**), C compiler (**-Tc**), linker (**-Tlk**) or locator (**-Tlc**). These options may be used to pass some options that are not recognized by the control program, to the appropriate program. The argument may be either directly appended to the option, or follow the option as a separate argument of the control program.

-V The copyright header containing the version number is displayed, after which the control program terminates.

-al Generate an absolute list file for each module in the application.

-c
-cl
-cs

Normally, the control program invokes all stages to build an absolute file from the given input files. With these options it is possible to skip the C compiler, assembler, linker or locator stage. With the **-cs** option the control program stops after the compilation of the C source files (`.c`) and after preprocessing the assembly source files (`.asm`), and retains the resulting `.src` files. With the **-c** option the control program stops after the assembler, with as output one or more object files (`.obj`). With the **-cl** option the control program stops after the link stage, with as output a linker object file (`.out`).

-f *file* Read command line arguments from *file*. The filename "-" may be used to denote standard input. To get around the limits on the size of the command line, it is possible to use command files. These command files contain the options that could not be part of the real command line. Command files can also be generated on the fly, for example by the make utility.
Some simple rules apply to the format of the command file:

1. It is possible to have multiple arguments on the same line in the command file.
2. To include whitespace in the argument, surround the argument with either single or double quotes.
3. If single or double quotes are to be used inside a quoted argument, we have to go by the following rules:
   a. If the embedded quotes are only single or double quotes, use the opposite quote around the argument. Thus, if a argument should contain a double quote, surround the argument with single quotes.
   b. If both types of quotes are used, we have to split the argument in such a way that each embedded quote is surrounded by the opposite type of quote.

   Example:

   ```
   "This has a single quote ' embedded"
   ```

   or

   ```
   'This has a double quote " embedded'
   ```

   or

   ```
   'This has a double quote " and a single quote '"' embedded"
   ```
4. Some operating systems impose limits on the length of lines within a text file. To circumvent this limitation it is possible to use continuation lines. These lines end with a backslash and newline. In a quoted argument, continuation lines will be appended without stripping any whitespace on the next line. For non-quoted arguments, all whitespace on the next line will be stripped.

   Example:

   ```
   "This is a continuation \
   line"
      → "This is a continuation line"

   control(file1(mode,type),\
      file2(type))
      → control(file1(mode,type),file2(type))
   ```
5. It is possible to nest command line files up to 25 levels.

-ieee
-srec With these options you can specify the locator output format of the absolute file. The output file can be an IEEE-695 file (`.abs`) or Motorola S-record file (`.sre`). The default output is IEEE-695 (`.abs`).

-nolib With this option the control program does not supply the standard libraries to the linker. Normally the control program supplies the default C and run-time libraries to the linker. Which libraries are needed is derived from the compiler options.

-o *file* Normally, this option is passed to the locator to specify the output file name. When you use the **-cl** option to suppress the locating phase, the **-o** option is passed to the linker. When you use the **-c** option to suppress the linking phase, the **-o** option is passed to the assembler, provided that only one source file is specified. When you use the **-cs** option to suppress the assembly phase, the **-o** option is passed to the compiler. The argument may be either directly appended to the option, or follow the option as a separate argument of the control program.

-tmp With this option the control program creates intermediate files in the current directory. They are not removed automatically. Normally, the control program generates temporary files for intermediate translation results, such as compiler generated assembly files, object files and the linker output file. If the next phase in the translation process completes successfully, these intermediate files will be removed.

-v When you use the **-v** option, the invocations of the individual programs are displayed on standard output, preceded by a '+' character.

-v0 This option has the same effect as the **-v** option, with the exception that only the invocations are displayed, but the programs are not started.

1.4.1.2 Environment Variables

The control program uses the following environment variables:

TMPDIR This variable may be used to specify a directory, which the control program should use to create temporary files. When this environment variable is not set, temporary files are created in the current directory.

CC88OPT This environment variable may be used to pass extra options and/or arguments to each invocation of the control program **cc88**. The control program processes the arguments from this variable before the command line arguments.

CC88BIN When this variable is set, the control program prepends the directory specified by this variable to the names of the tools invoked.

1.4.2 Compiler

The invocation syntax of the C compiler is:

c88 [[*option*] ... [*file*] ...] ...

The C compiler accepts C source file names and command line options in random order. Source files are processed in the same order as they appear on the command line (left-to-right). Options are indicated by a leading '-' character. Each C source file is compiled separately and the compiler generates an output file with suffix `.src` per C source module, containing assembly source code.

The priority of the options is left-to-right: when two options conflict, the first (most left) one takes effect. You can overrule the default output file name with the **-o** option. The compiler uses each **-o** option only once, so it is possible to specify multiple **-o** options for multiple source files.

A summary of the options is given below. The next section describes the options in more detail.

Compiler Options

Option	Description
-D*macro*[=*def*]	Define preprocessor *macro*
-H *file*	Include *file* before starting compilation
-I*directory*	Look in *directory* for include files
-M{**s**|**c**|**d**|**l**}	Select memory model: small, compact code, compact data or large
-O{**0**|**1**}	Control optimization
-V	Display version header only
-e	Remove output file if compiler errors occur
-err	Send diagnostics to error list file (`.err`)
-f *file*	Read options from *file*
-g	Enable symbolic debug information
-o *file*	Specify name of output *file*
-s	Merge C-source code with assembly output
-w[*num*]	Suppress one or all warning messages

Compiler Options (functional order)

Description	Option
Include options	
Read options from *file*	**-f** *file*
Include *file* before starting compilation	**-H** *file*
Look in *directory* for include files	**-I***directory*
Preprocess options	
Define preprocessor *macro*	**-D***macro*[=*def*]
Code generation options	
Select memory model: small, compact code, compact data or large	**-M**{**s**|**c**|**d**|**l**}
Control optimization	**-O**{**0**|**1**}
Output file options	
Remove output file if compiler errors occur	**-e**
Specify name of output *file*	**-o** *file*
Merge C-source code with assembly output	**-s**
Diagnostic options	
Display version header only	**-V**
Send diagnostics to error list file (`.err`)	**-err**
Enable symbolic debug information	**-g**
Suppress one or all warning messages	**-w**[*num*]

1.4.2.1 Detailed Description of the Compiler Options

Option letters are listed below. Each option (except **-o**; see description of the **-o** option) is applied to every source file. If the same option is used more than once, the first (most left) occurrence is used. The placement of command line options is of no importance except for the **-I** and **-o** options. For the **-o** option, the filename may not start immediately after the option. There must be a tab or space in between. All other option arguments must start immediately after the option. Source files are processed in the same order as they appear on the command line (left-to-right).

Some options have an equivalent pragma.

-D

Option:

-D*macro*[*=def*]

Arguments:

The macro you want to define and optionally its definition.

Description:

Define *macro* to the preprocessor, as in #define. If *def* is not given ('=' is absent), '1' is assumed. Any number of symbols can be defined. The definition can be tested by the preprocessor with #if, #ifdef and #ifndef, for conditional compilations. If the command line is getting longer than the limit of the operating system used, the **-f** option is needed.

ANSI specifies the following predefined symbols to exist, which cannot be removed:

__FILE__ "current source filename"

__LINE__ current source line number (int type)

__TIME__ "hh:mm:ss"

__DATE__ "Mmm dd yyyy"

__STDC__ level of ANSI standard. This macro is set to 0 (zero).

When **c88** is invoked, also the following predefined symbols exist:

_C88 predefined symbol to identify the compiler. This symbol can be used to flag parts of the source which must be recognized by the **c88** compiler only. It expands to the version number of the compiler.

_MODEL identifies for which memory model the module is compiled. It expands to a single character ('t' for tiny, 's' for small, 'm' for medium or 'l' for large) that can be tested by the preprocessor. See Section 1.2.2.2, "Memory Models" for details.

Example:

The following command defines the symbol `NORAM` as `1` and defines the symbol `PI` as `3.1416`.

```
c88 -DNORAM -DPI=3.1416 test.c
```

-e

Option:

-e

Description:

Remove the output file when an error has occurred. With this option the 'make' utility always does the proper productions.

Example:

```
c88 -e test.c
```

-err

Option:

-err

Description:

Write errors to the file *source*.err instead of stderr.

Example:

To write errors to the test.err instead of stderr, enter:

```
c88 -err test.c
```

-f

Option:

-f *file*

Arguments:

A filename for command line processing. The filename "-" may be used to denote standard input.

Description:

Use *file* for command line processing. To get around the limits on the size of the command line, it is possible to use command files. These command files contain the options that could not be part of the real command line. Command files can also be generated on the fly, for example by the make utility. More than one **-f** option is allowed.
Some simple rules apply to the format of the command file:

1. It is possible to have multiple arguments on the same line in the command file.
2. To include whitespace in the argument, surround the argument with either single or double quotes.
3. If single or double quotes are to be used inside a quoted argument, we have to go by the following rules:
   a. If the embedded quotes are only single or double quotes, use the opposite quote around the argument. Thus, if a argument should contain a double quote, surround the argument with single quotes.
   b. If both types of quotes are used, we have to split the argument in such a way that each embedded quote is surrounded by the opposite type of quote.

   Example:

   ```
   "This has a single quote ' embedded"
   ```

   or

   ```
   'This has a double quote " embedded'
   ```

   or

   ```
   'This has a double quote " and a single quote '"' embedded"
   ```
4. Some operating systems impose limits on the length of lines within a text file. To circumvent this limitation it is possible to use continuation lines. These lines end with a backslash and newline. In a quoted argument, continuation lines will be appended without stripping any whitespace on the next line. For non-quoted arguments, all whitespace on the next line will be stripped.

   Example:

   ```
   "This is a continuation \
   line"
      → "This is a continuation line"

   control(file1(mode,type),\
      file2(type))
      → control(file1(mode,type),file2(type))
   ```
5. It is possible to nest command line files up to 25 levels.

Example:

Suppose the file `mycmds` contains the following lines:

```
-err
test.c
```

The command line can now be:

```
c88 -f mycmds
```

-g

Option:

-g

Description:

Add directives to the output files, incorporating symbolic information to facilitate high level debugging.

When the compiler is set to a high optimization level the debug comfort may decrease.

Examples:

To add symbolic debug information to the output files, enter:

```
c88 -g test.c
```

See also:

-O

-H

Option:

-H*file*

Arguments:

The name of an include file.

Description:

Include *file* before compiling the C-source. This is the same as specifying #include "*file*" at the first line of your C-source.

Example:

```
c88 -Hstdio.h test.c
```

See also:

-I

-I

Option:

-I*directory*

Arguments:

The name of the directory to search for include file(s).

Description:

Change the algorithm for searching #include files whose names do not have an absolute pathname to look in *directory*. Thus, #include files whose names are enclosed in "" are searched for first in the directory of the file containing the #include line, then in directories named in **-I** options in left-to-right order. If the include file is still not found, the compiler searches in a directory specified with the environment variable C88INC. C88INC may contain more than one directory.

Finally, the directory `../include` relative to the directory where the compiler binary is located is searched. This is the standard include directory supplied with the compiler package.

For #include files whose names are in <>, the directory of the file containing the #include line is not searched. However, the directories named in **-I** options (and the one in C88INC and the relative path) are still searched.

Example:

```
c88 -I/proj/include test.c
```

See also:

Section 1.4.3, "Include Files".

-M

Option:

-M*model*

Arguments:

The memory model to be used, where *model* is one of:

s small
c compact code
d compact data
l large

Default:

-Ms

Description:

Select memory model to be used.

Example:

```
c88 -Ml test.c
```

See also:

Section 1.2.2.2, "Memory Models".

-O

Option:

-O*flag*

Arguments:

0 or **1**

Default:

-O1

Description:

Control optimization. You can specify a single number 1 or 0, to enable or disable optimization.

-O0 - Switchable optimizations switched off.

-O1 - Default. Set optimization to let **c88** generate the smallest code.

An overview of the optimization using the **-O** option is given below.

Relax alias checking

With **-O1** you relax alias checking. If you specify this option, **c88** will not erase remembered register contents of user variables if a write operation is done via an indirect (calculated) address. You must be sure this is not done in your C-code (check pointers!) before turning on this option.

With **-O0** you specify strict alias checking. If you specify this option, the compiler erases all register contents of user variables when a write operation is done via an indirect (calculated) address.

Clearing of non-initialized static and public variables

The compiler performs 'clearing' of non-initialized static and public variables regardless of the option specified.

Common subexpression elimination

With **-O1** you enable CSE (common subexpression elimination). With this option specified, the compiler tries to detect common subexpressions within the C code. The common expressions are evaluated only once, and their result is temporarily held in registers.

With **-O0** you disable CSE (common subexpression elimination). With this option specified, the compiler will not try to search for common expressions. Also relax alias checking, expression propagation and moving invariant code outside a loop will be disabled.

Example:

```
/*
 * Compile with -O0,
 * Compile with -O1, common subexpressions are found
 *      and temporarily saved.
 */

char x, y, a, b, c, d;

void
main( void )
{
        x = (a * b) - (c * d);
        y = (a * b) + (c * d);/*(a*b) and (c*d) are common */
}
```

Data flow, constant/copy propagation

With **-O1** you enable constant and copy propagation. With this option, the compiler tries to find assignments of constant values to a variable, a subsequent assignment of the variable to another variable can be replaced by the constant value.

With **-O0** you disable constant and copy propagation.

Example:

```
/*
 * Compile with -O0, 'i' is actually assigned to 'j'
 * Compile with -O1, 15 is assigned to 'j', 'i' was
 * propagated
 */

int i;
int j;

void
main( void )
{
        i = 10;
        j = i + 5;
}
```

Expression propagation

With **-O1** you enable expression propagation. With this option, the compiler tries to find assignments of expressions to a variable, a subsequent assignment of the variable to another variable can be replaced by the expression itself.

With **-O0** you disable expression propagation.

Example:

```
/*
 * Compile with -O0, normal cse is done
 * Compile with -O1, 'i+j' is propagated.
 */

unsigned i, j;

int
main( void )
{
        static int a;
        a = i + j;
        return (a);
}
```

Code flow, order rearranging

With **-O1** you enable control flow optimizations and code order rearranging on the intermediate code representation, such as jump chaining and conditional jump reversal.

With **-O0** you disable control flow optimizations.

Examples:

The following example shows a control optimization:

```
/*
 * Compile with -O0
 * Compile with -O1, compiler finds first time 'i' is
 * always < 10, the unconditional jump is removed.
 */
int i;

void
main( void )
{
        for( i=0; i<10; i++ )
        {
                do_something();
        }
}
```

The following example shows a conditional jump reversal:

```
/*
 * Compile with -O0, code as written sequential
 * Compile with -O1, code is rearranged
 * Code rearranging enables other optimizations to optimize better, e.g. CSE
 */

int     i;
extern  void    dummy( void );

void main ()
{
        do
        {
                if      ( i )
                {
                        i--;
                }
                else
                {
                        i++;
                        break;
                }
                dummy();
        } while ( i );
}
```

Peephole optimization

With **-O1** you enable peephole optimization. Remove redundant code. The peephole optimizer searches for redundant instructions or for instruction sequences which can be combined to minimize the number of instructions.

With **-O0** you disable peephole optimization.

Move invariant code outside loop

With **-O1** you move invariant code outside a loop.

With **-O0** you disable moving invariant code outside a loop.

Example:

```
/*
 * Compile with -OI -Oc -O0, normal cse is done
 * Compile with -Oi -Oc -O0, invariant code is found in
 *      the loop, code is moved outside the loop.
 */
void
main( void )
{
        char x, y, a, b;
        int i;

        for( i=0; i<20; i++ )
        {
                x = a + b;
                y = a + b;
        }
}
```

Fast loops (increases code size)

Fast loops are disabled regardless of the option specified.

Small code size

With **-O1** you tell the compiler to generate smaller code. Whenever possible less instructions are used. Note that this may result in more instruction cycles.

With **-O0** you disable the smaller code optimization.

Loop unrolling

Loop unrolling is disabled regardless of the option specified.

Subscript strength reduction

With **-O1** you enable subscript strength reduction. With this option specified, the compiler tries to reduce expressions involving an index variable in strength.

With **-O0** you disable subscript strength reduction.

Example:

```
/*
 * Compile with -O0, disable subscript strength reduction
 * Compile with -O1, begin and end address of 'a' are
 * determined before the loop and temporarily put in registers
 * instead of determining the address each time inside the loop
 */
int i;
int a[4];

void
main( void )
{
        for( i=0; i<4; i++ )
        {
                a[i] = i;
        }
}
```

-O

Option:

-o *file*

Arguments:

An output filename. The filename may not start immediately after the option. There must be a tab or space in between.

Default:

Module name with `.src` suffix.

Description:

Use *file* as output filename, instead of the module name with `.src` suffix. Special care must be taken when using this option, the first **-o** option found acts on the first file to compile, the second **-o** option acts on the second file to compile, etc.

Example:

When specified:

```
c88 file1.c file2.c -o file3.src -o file2.src
```

two files will be created, `file3.src` for the compiled file `file1.c` and `file2.src` for the compiled file `file2.c`.

-S

Option:

-s

Pragma:

source

Description:

Merge C source code with generated assembly code in output file.

Example:

```
c88 -s test.c
;             test.c:
; 1    |int i;
; 2    |
; 3    |int
; 4    |main( void )
; 5    |{
        extern  __START
        global  _main
```

See also:

Pragmas `source` and `nosource` in Section 1.4.4, "Pragmas".

-V

Option:

-V

Description:

Display version information.

Example:

```
c88 -V
S1C88 C compiler vx.y rz     SN00000000-015 (c) year TASKING, Inc.
```

-W

Option:

-w[*num*]

Arguments:

Optionally the warning number to suppress.

Description:

-w suppress all warning messages. **-w***num* only suppresses the given warning.

Example:

To suppress warning 135, enter:

```
c88 file1.c -w135
```

1.4.3 Include Files

You may specify include files in two ways: enclosed in <> or enclosed in "". When an #include directive is seen, **c88** used the following algorithm trying to open the include file:

1. If the filename is enclosed in "", and it is not an absolute pathname (does not begin with a '\', the include file is searched for in the directory of the file containing the #include line. For example, in:

   ```
   c88 ..\..\source\test.c
   ```

   c88 first searches in the directory `..\..\source` for include files.

   If you compile a source file in the directory where the file is located (**c88 test.c**), the compiler searches for include files in the current directory.

   Note that this first step is not done for include files enclosed in <>.

2. Use the directories specified with the **-I** options, in a left-to-right order. For example:

   ```
   c88 -I..\..\include demo.c
   ```

3. Check if the environment variable C88INC exists. If it does exist, use the contents as a directory specifier for include files. You can specify more than one directory in the environment variable C88INC by using a separator character. Instead of using **-I** as in the example above, you can specify the same directory using C88INC:

   ```
   set C88INC=..\..\include
   c88 demo.c
   ```

4. When an include file is not found with the rules mentioned above, the compiler tries the subdirectory `include`, one directory higher than the directory containing the **c88** binary. For example:

   c88.exe is installed in the directory `C:\C88\BIN`
   The directory searched for the include file is `C:\C88\INCLUDE`

   The compiler determines run-time which directory the binary is executed from to find this `include` directory.

A directory name specified with the **-I** option or in C88INC may or may not be terminated with a directory separator, because **c88** inserts this separator, if omitted.

When you specify more than one directory to the environment variable C88INC, you have to use one of the following separator characters:

`;` `,` *space*

e.g. `set C88INC=..\..\include;\proj\include`

1.4.4 Pragmas

According to ANSI (3.8.6) a preprocessing directive of the form:

```
#pragma pragma-token-list new-line
```

causes the compiler to behave in an implementation-defined manner. The compiler ignores pragmas which are not mentioned in the list below. Pragmas give directions to the code generator of the compiler. Besides the pragmas there are two other possibilities to steer the code generation process: command line options and keywords (e.g., `_near` type variables) in the C application itself. The compiler acknowledges these three groups using the following rules:

Command line options can be overruled by keywords and pragmas. Keywords can be overruled by pragmas. Hence, pragmas have the highest priority.

This approach makes it possible to set a default optimization level for a source module, which can be overridden temporarily within the source by a pragma.

The C compiler **c88** supports the following pragmas:

asm

Insert the following (non preprocessor lines) as assembly language source code into the output file. The inserted lines are not checked for their syntax. The code buffer of the peephole optimizer is flushed. Thus the compiler will stop optimizations like peephole pattern replacement and resumes these optimizations after the **endasm** pragma as if it starts at the beginning of a function.

For advanced assembly in-lining, intrinsic functions can be used. The defined set of intrinsic functions cover most of the specific S1C88 features which could otherwise not be accessed by the C language. For more information on intrinsic functions see Section 1.2.16, "Intrinsic Functions".

asm_noflush

Same as pragma **asm**, except that the peephole optimizer does not flush the code buffer and assumes register contents remain valid.

endasm

Switch back to the C language.
Section 1.2.14, "Inline Assembly", contains more information.

source

Same as **-s** option. Enable mixing C source with assembly code.

nosource

Default. Disable generation of C source within assembly code.

1.4.5 Compiler Limits

The ANSI C standard [1-2.2.4] defines a number of translation limits, which a C compiler must support to conform to the standard. The standard states that a compiler implementation should be able to translate and execute a program that contains at least one instance of every one of the limits listed below. **c88**'s actual limits are given within parentheses.

Most of the actual compiler limits are determined by the amount of free memory in the host system. In this case a 'D' (Dynamic) is given between parentheses. Some limits are determined by the size of the internal compiler parser stack. These limits are marked with a 'P'. Although the size of this stack is 200, the actual limit can be lower and depends on the structure of the translated program.

- 15 nesting levels of compound statements, iteration control structures and selection control structures (P > 15)
- 8 nesting levels of conditional inclusion (50)
- 12 pointer, array, and function declarators (in any combinations) modifying an arithmetic, a structure, a union, or an incomplete type in a declaration (15)
- 31 nesting levels of parenthesized declarators within a full declarator (P > 31)
- 32 nesting levels of parenthesized expressions within a full expression (P > 32)
- 31 significant characters in an external identifier (full ANSI-C mode),
  500 significant characters in an external identifier (non ANSI-C mode)
- 511 external identifiers in one translation unit (D)
- 127 identifiers with block scope declared in one block (D)
- 1024 macro identifiers simultaneously defined in one translation unit (D)
- 31 parameters in one function declaration (D)
- 31 arguments in one function call (D)
- 31 parameters in one macro definition (D)
- 31 arguments in one macro call (D)
- 509 characters in a logical source line (1500)
- 509 characters in a character string literal or wide string literal (after concatenation) (1500)
- 8 nesting levels for **#includ**ed files (50)
- 257 case labels for a switch statement, excluding those for any nested switch statements (D)
- 127 members in a single structure or union (D)
- 127 enumeration constants in a single enumeration (D)
- 15 levels of nested structure or union definitions in a single struct-declaration-list (D)

1.4.6 Linker Messages

c88 has three classes of messages: user errors, warnings and internal compiler errors.

Some user error messages carry extra information, which is displayed by the compiler after the normal message. The messages with extra information are marked with 'I' in the list described in Appendix. They never appear without a previous error message and error number. The number of the information message is not important, and therefore, this number is not displayed. A user error can also be fatal (marked as 'F' in the list described in Appendix), which means that the compiler aborts compilation immediately after displaying the error message and may generate a 'not complete' output file.

The error numbers and warning numbers are divided in two groups. The frontend part of the compiler uses numbers in the range 0 to 499, whereas the backend (code generator) part of the compiler uses numbers in the range 500 and higher. Note that most error messages and warning messages are produced by the frontend.

If you program a non fatal error, **c88** displays the C source line that contains the error, the error number and the error message on the screen. If the error is generated by the code generator, the C source line displayed always is the last line of the current C function, because code generation is started when the end of the function is reached by the frontend. However, in this case, **c88** displays the line number causing the error before the error message. **c88** always generates the error number in the assembly output file, exactly matching the place where the error occurred.

So, when a compilation is not successful, the generated output file is not accepted by the assembler, thus preventing a corrupt application to be made (see also the **-e** option).

Warning messages do not result into an erroneous assembly output file. They are meant to draw your attention to assumptions of the compiler, for a situation which may not be correct. Warning messages can be controlled with the **-w**[*num*] option.

The last class of messages are the internal compiler errors. The following format is used:

```
S number: internal error - please report
```

These errors are caused by failed internal consistency checks and should never occur. However, if such a 'SYSTEM' error appears, please report the occurrence to Seiko Epson. Please include a diskette or tape, containing a small C program causing the error.

1.4.7 Return Values

c88 returns an exit status to the operating system environment for testing.

For example,

in a MS-DOS BATCH-file you can examine the exit status of the program executed with ERRORLEVEL:

```
c88 -s %1.c
IF ERRORLEVEL 1 GOTO STOP_BATCH
```

The exit status of **c88** is one of the numbers of the following list:

Exit status:

0 Compilation successful, no errors

1 There were user errors, but terminated normally

2 A fatal error, or System error occurred, premature ending

3 Stopped due to user abort

1.5 Libraries

This chapter describes the library functions delivered with the compiler. Some functions (e.g. `printf()`, `scanf()`) can be edited to match your needs. **c88** come with libraries in object format per memory model and with header files containing the appropriate prototype of the library functions. The library functions are also shipped in source code (C or assembly).

A number of standard operations within C are too complex to generate inline code for (e.g. 32 bit signed divide). These operations are implemented as run-time library functions.

Note: Use the library functions at on the user's own risk after performing enough evaluation, since the function operations cannot be guaranteed.

1.5.1 Header Files

The following header files are delivered with the C compiler:

<assert.h> assert

<c88.h> Special file with **c88** definitions. No C functions. Can be used for prototyping your application on a host using a standard C compiler.

<ctype.h> isalnum, isalpha, isascii, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, toascii, _tolower, tolower, _toupper, toupper

<errno.h> Error numbers. No C functions.

<limits.h> Limits and sizes of integral types. No C functions.

<locale.h> localeconv, setlocale. Delivered as skeletons.

<setjmp.h> longjmp, setjmp

<signal.h> raise, signal. Functions are delivered as skeletons.

<stdarg.h> va_arg, va_end, va_start

<stddef.h> offsetof, definition of special types.

<stdio.h> clearerr, fclose, _fclose, feof, ferror, fflush, fgetc, fgetpos, fgets, fopen, _fopen, fprintf, fputc, fputs, fread, freopen, fscanf, fseek, fsetpos, ftell, fwrite, getc, getchar, gets, _ioread, _iowrite, _lseek, perror, printf, putc, putchar, puts, _read, remove, rename, rewind, scanf, setbuf, setvbuf, sprintf, sscanf, tmpfile, tmpnam, ungetc, vfprintf, vprintf, vsprintf, _write

<stdlib.h> abort, abs, atexit, atof, atoi, atol, bsearch, calloc, div, exit, free, getenv, labs, ldiv, malloc, mblen, mbstowcs, mbtowc, qsort, rand, realloc, srand, strtod, strtol, strtoul, system, wcstombs, wctomb

<string.h> memchr, memcmp, memcpy, memmove, memset, strcat, strchr, strcmp, strcol, strcpy, strcspn, strerror, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strstr, strtok, strxfrm

<time.h> asctime, clock, ctime, difftime, gmtime, localtime, mktime, strftime, time. All functions are delivered as skeletons.

1.5.2 C Libraries

The C library contains C library functions. All C library functions are described in this chapter. These functions are only called by explicit function calls in your application program.

The `lib` directory contains subdirectories for the different processor types. The C library uses the following name syntax:

Table 1.5.2.1 C library name syntax

Compiler Model	Library to link
Small (default)	libcs.a (default)
Compact code	libcc.a
Compact data	libcd.a
Large	libcl.a

Note that the **lk88** linker is using this naming convention when specifying the **-l** option. For example, with **-lcd** the linker is looking for `libcd.a` in the system `lib` directory. Specifying the libraries is a job taken care of by the control program.

1.5.2.1 C Library Implementation Details

A detailed description of the delivered C library is shown in the following list.

Some C library routines need to be recompiled before they can be used from a program. These library functions are not activated by default, because of the extra memory required.

Explanation:

- Y - Fully implemented
- I - Implemented, but need some user written low level routine
- R - Implemented, but needs recompilation
- L - Delivered as a skeleton

File	Implemented	Routine name	Description / Reason
assert.h	Y	'assert()' macro	Macro definition
ctype.h	Y		Most of the routines are delivered as macro AND as function (as prescribed by ANSI).
	Y	isalnum	
	Y	isalpha	
	Y	iscntrl	
	Y	isdigit	
	Y	isgraph	
	Y	islower	
	Y	isprint	
	Y	ispunct	
	Y	isspace	
	Y	isupper	
	Y	isxdigit	
	Y	tolower	
	Y	toupper	
	Y	_tolower	Not defined by ANSI
	Y	_toupper	Not defined by ANSI
	Y	isascii	Not defined by ANSI
	Y	toascii	Not defined by ANSI
errno.h	Y		Only Macros
limits.h	Y		Only Macros
locale.h	Y		
	L	localeconv	No OS present
	L	setlocale	No OS present

File	Implemented	Routine name	Description / Reason
setjmp.h	Y		
	Y	longjmp	
	Y	setjmp	
signal.h	Y		
	L	raise	No OS present
	L	signal	No OS present
stdarg.h	Y		
	Y	va_arg	
	Y	va_end	
	Y	va_start	
stddef.h	Y		Only Macros
stdio.h	Y		
	Y	clearerr	
	I	fclose	Needs _fclose
	Y	feof	
	Y	ferror	
	I	fflush	
	I	fgetc	Needs _write/_iowrite
	I	fgetpos	Needs _read/_ioread
	I	fgets	Needs _lseek
	I	fopen	Needs _read/_ioread
	I	fprintf	Needs _fopen
	I	fputc	Needs _write/_iowrite
	I	fputs	Needs _write/_iowrite
	I	fread	Needs _write/_iowrite
	I	freopen	Needs _read/_ioread
	I	fscanf	Needs _fclose/_fopen
	I	fseek	Needs _read/_ioread
	I	fsetpos	Needs _lseek
	I	ftell	Needs _lseek
	I	fwrite	Needs _lseek
	I	getc	Needs _write/_iowrite
	I	getchar	Needs _read/_ioread
	I	gets	Needs _read/_ioread
	I	perror	Needs _write/_iowrite
	I	printf	Needs _write/_iowrite
	I	putc	Needs _write/_iowrite
	I	putchar	Needs _write/_iowrite
	I	puts	Needs _write/_iowrite
	L	remove	
	L	rename	
	I	rewind	Needs _lseek
	I	scanf	Needs _read/_ioread
	Y	setbuf	
	Y	setvbuf	
	Y	sprintf	
	Y	sscanf	
	L	tmpfile	
	L	tmpnam	Delivered as a random name generator, but should use some process ID.
	Y	ungetc	
	I	vfprintf	Needs _write/_iowrite
	I	vprintf	Needs _write/_iowrite
	Y	vsprintf	
	L	_fclose	Low level file close routine
	L	_fopen	Low level file open routine
	L	_ioread	Low level input routine
	L	_iowrite	Low level output routine
	L	_lseek	Low level file positioning routine
	L	_read	Low level block input routine, when not customized, will use _ioread
	L	_write	Low level block write routine, when not customized, will use _iowrite

File	Implemented	Routine name	Description / Reason
stdlib.h	Y		
	Y	abort	Calls _exit() in cstart
	Y	abs	
	R	atexit	Needs recompilation of exit()
	Y	atoi	
	Y	atol	
	Y	bsearch	
	Y	calloc	
	Y	div	
	Y	exit	Calls _exit() in cstart
	Y	free	
	L	getenv	No OS present
	Y	labs	
	Y	ldiv	
	Y	malloc	
	Y	qsort	
	Y	strtod	
	Y	strtol	
	Y	strtoul	
	Y	rand	
	Y	realloc	
	Y	srand	
	L	system	No OS present
	L	mblen	wide chars not supported
	L	mbstowcs	wide chars not supported
	L	mbtowc	wide chars not supported
	L	wcstombs	wide chars not supported
	L	wctomb	wide chars not supported
string.h	Y		
	Y	memchr	
	Y	memcmp	
	Y	memcpy	
	Y	memmove	
	Y	memset	
	Y	strcat	
	Y	strchr	
	Y	strcmp	
	L	strcoll	wide chars not supported
	Y	strcpy	
	Y	strcspn	
	Y	strerror	
	Y	strlen	
	Y	strncat	
	Y	strncmp	
	Y	strncpy	
	Y	strpbrk	
	Y	strrchr	
	Y	strspn	
	Y	strstr	
	Y	strtok	
	L	strxfrm	wide chars not supported
time.h	Y		real time clock not supported
	L	asctime	
	L	clock	
	L	ctime	
	L	gmtime	
	L	localtime	
	L	mktime	
	L	strftime	
	L	time	

1.5.2.2 C Library Interface Description

_fclose

```
#include <stdio.h>
int _fclose( FILE *file );
```

Low level file close function. _fclose is used by the function fclose. The given stream should be properly closed, any buffer is already flushed.

_fopen

```
#include <stdio.h>
int _fopen( const char *file, FILE *iop );
```

Low level file open function. _fopen is used by the functions fopen and freopen. The given stream should be properly opened.

_ioread

```
#include <stdio.h>
int _ioread( FILE *fp );
```

Low level input function. The delivered library contains an 'empty' function. To perform real I/O, you must customize this function. _ioread is used by all input functions (scanf, getc, gets, etc.).

_iowrite

```
#include <stdio.h>
int _iowrite( int c, FILE *fp );
```

Low level output function. The delivered library contains an 'empty' function. To perform real I/O, you must customize this function. _iowrite is used by all output functions (printf, putc, puts, etc.).

_lseek

```
#include <stdio.h>
long _lseek( FILE *iop, long offset, int origin );
```

Low level file positioning function. _lseek is used by all file positioning functions (fgetpos, fseek, fsetpos, ftell, rewind).

_read

```
#include <stdio.h>
size_t _read( FILE *fin, char *base, size_t size );
```

Low level block input function. You must customize this function before using it. When not customized it will use _ioread. It is used by all input functions. It reads a block of characters from the given stream.

Returns the number of characters read.

_tolower

```
#include <ctype.h>
int _tolower( int c );
```

Converts `c` to a lowercase character, does not check if `c` really is an uppercase character. This is a non-ANSI function.

Returns the converted character.

_toupper

```
#include <ctype.h>
int _toupper( int c );
```

Converts `c` to an uppercase character, does not check if `c` really is a lowercase character. This is a non-ANSI function.

Returns the converted character.

_write

```
#include <stdio.h>
size_t _write( FILE *iop, char *base, size_t size );
```

Low level block output function. You must customize this function before using it. When not customized it will use _iowrite. It is used by all output functions. It writes a block of characters to the given stream.

Returns the number of characters correctly written.

abort

```
#include <stdlib.h>
void abort( void );
```

Terminates the program abnormally. It calls the function `_exit`, which is defined in the start-up module.

Returns nothing.

abs

```
#include <stdlib.h>
int abs( int n );
```

Returns the absolute value of the signed int argument.

asctime

```
#include <time.h>
char *asctime( const struct tm *tp );
```

Converts the time in the structure `*tp` into a string of the form:

`Mon Jan 21 16:15:14 1989\n\0`

Returns the time in string form.

assert

```
#include <assert.h>
void assert( int expr );
```

When compiled with NDEBUG, this is an empty macro. When compiled without NDEBUG defined, it checks if `expr` is true. If it is true, then a line like:

"Assertion failed: expression, file *filename*, line *num*"

is printed.

Returns nothing.

atexit

```
#include <stdlib.h>
int atexit( void (*fcn)( void ) );
```

Registers the function `fcn` to be called when the program terminates normally.

Returns zero, if program terminates normally. non-zero, if the registration cannot be made.

atoi

```
#include <stdlib.h>
int atoi( const char *s );
```

Converts the given string to an integer value. White space is skipped, conversion is terminated at the first unrecognized character.

Returns the integer value.

atol

```
#include <stdlib.h>
long atol( const char *s );
```

Converts the given string to a long value. White space is skipped, conversion is terminated at the first unrecognized character.

Returns the long value.

bsearch

```
#include <stdlib.h>
void *bsearch( const void *key, const void *base, size_t n, size_t size,
  int (* cmp) (const void *, const void *) );
```

This function searches in an array of `n` members, for the object pointed to by `ptr`. The initial base of the array is given by `base`. The size of each member is specified by `size`. The given array must be sorted in ascending order, according to the results of the function pointed to by `cmp`.

Returns a pointer to the matching member in the array, or NULL when not found.

calloc

```
#include <stdlib.h>
void *calloc( size_t nobj, size_t size );
```

The allocated space is filled with zeros. The maximum space that can be allocated can be changed by customizing the heap size (see Section 1.3.5, "Heap"). By default no heap is allocated. When "calloc()" is used while no heap is defined, the locator gives an error.

Returns a pointer to space in external memory for `nobj` items of `size` bytes length.
NULL if there is not enough space left.

clearerr

```
#include <stdio.h>
void clearerr( FILE *stream );
```

Clears the end of file and error indicators for stream.

Returns nothing.

clock

```
#include <time.h>
clock_t clock( void );
```

Determines the processor time used.

Returns 1.

ctime

```
#include <time.h>
char *ctime( const time_t *tp );
```

Converts the calender time `*tp` into local time, in string form. This function is the same as:

```
asctime( localtime( tp ) );
```

Returns the local time in string form.

div

```
#include <stdlib.h>
div_t div( int num, int denom );
```

Both arguments are integers. The returned quotient and remainder are also integers.

Returns a structure containing the quotient and remainder of `num` divided by `denom`.

exit

```
#include <stdlib.h>
void exit( int status );
```

Terminates the program normally. Acts as if 'main()' returns with `status` as the return value.

Returns zero, on successful termination.

fclose

```
#include <stdio.h>
int fclose( FILE *stream )
```

Flushes any unwritten data for stream, discards any unread buffered input, frees any automatically allocated buffer, then closes the `stream`.

Returns zero if the `stream` is successfully closed, or EOF on error.

feof

```
#include <stdio.h>
int feof( FILE *stream );
```

Returns a non-zero value if the end-of-file indicator for `stream` is set.

ferror

```
#include <stdio.h>
int ferror( FILE *stream );
```

Returns a non-zero value if the error indicator for `stream` is set.

fflush

```
#include <stdio.h>
int fflush( FILE *stream );
```

Writes any buffered but unwritten date, if `stream` is an output stream. If `stream` is an input stream, the effect is undefined.

Returns zero if successful, or EOF on a write error.

fgetc

```
#include <stdio.h>
int fgetc( FILE *stream );
```

Reads one character from the given `stream`.

Returns the read character, or EOF on error.

fgetpos

```
#include <stdio.h>
int fgetpos( FILE *stream, fpos_t *ptr );
```

Stores the current value of the file position indicator for the stream pointed to by `stream` in the object pointed to by `ptr`. The type `fpos_t` is suitable for recording such values.

Returns zero if successful, a non-zero value on error.

fgets

```
#include <stdio.h>
char *fgets( char *s, int n, FILE *stream );
```

Reads at most the next `n`-1 characters from the given `stream` into the array `s` until a newline is found.

Returns `s`, or NULL on EOF or error.

fopen

```
#include <stdio.h>
FILE *fopen( const char *filename, const char *mode );
```

Opens a file for a given `mode`.

Returns a stream. If the file cannot not be opened, NULL is returned.

You can specify the following values for `mode`:

"r"	read; open text file for reading
"w"	write; create text file for writing; if the file already exists its contents is discarded
"a"	append; open existing text file or create new text file for writing at end of file
"r+"	open text file for update; reading and writing
"w+"	create text file for update; previous contents if any is discarded
"a+"	append; open or create text file for update, writes at end of file

The update mode (with a '+') allows reading and writing of the same file. In this mode the function fflush must be called between a read and a write or vice versa. By including the letter b after the initial letter, you can indicate that the file is a binary file. E.g. "rb" means read binary, "w+b" means create binary file for update. The filename is limited to FILENAME_MAX characters. At most FOPEN_MAX files may be open at once.

fprintf

```
#include <stdio.h>
int fprintf( FILE *stream, const char *format, ... );
```

Performs a formatted write to the given `stream`. See also "printf()", "_iowrite()" and Section 1.5.2.3, "Printf and Scanf Formatting Routines".

fputc

```
#include <stdio.h>
int fputc( int c, FILE *stream );
```

Puts one character onto the given `stream`. See also "_iowrite()".

Returns EOF on error.

fputs

```
#include <stdio.h>
int fputs( const char *s,  FILE *stream );
```

Writes the string to a `stream`. The terminating NULL character is not written. See also "_iowrite()".

Returns NULL if successful, or EOF on error.

fread

```
#include <stdio.h>
size_t fread( void *ptr, size_t size, size_t nobj, FILE *stream );
```

Reads `nobj` members of `size` bytes from the given `stream` into the array pointed to by `ptr`. See also "_ioread()".

Returns the number of successfully read objects.

free

```
#include <stdlib.h>
void free( void *p );
```

Deallocates the space pointed to by `p`. `p` must point to space earlier allocated by a call to "calloc()", "malloc()" or "realloc()". Otherwise the behavior is undefined. See also "calloc()", "malloc()" and "realloc()".

Returns nothing.

freopen

```
#include <stdio.h>
FILE *freopen( const char *filename, const char *mode, FILE *stream );
```

Opens a file for a given `mode` associates the `stream` with it. This function is normally used to change the files associated with stdin, stdout, or stderr. See also "fopen()".

Returns `stream`, or NULL on error.

fscanf

```
#include <stdio.h>
int fscanf( FILE *stream, const char *format, ... );
```

Performs a formatted read from the given `stream`. See also "scanf()", "_ioread()" and Section 1.5.2.3, "Printf and Scanf Formatting Routines".

Returns the number of items converted successfully.

fseek

```
#include <stdio.h>
int fseek( FILE *stream, long offset, int origin );
```

Sets the file position indicator for `stream`. A subsequent read or write will access data beginning at the new position. For a binary file, the position is set to `offset` characters from `origin`, which may be SEEK_SET for the beginning of the file, SEEK_CUR for the current position in the file, or SEEK_END for the end-of-file. For a text stream, `offset` must be zero, or a value returned by `ftell`. In this case `origin` must be SEEK_SET.

Returns zero if successful, a non-zero value on error.

fsetpos

```
#include <stdio.h>
int fsetpos( FILE *stream, const fpos_t *ptr );
```

Positions `stream` at the position recorded by `fgetpos` in `*ptr`.

Returns zero if successful, a non-zero value on error.

ftell

```
#include <stdio.h>
long ftell( FILE *stream );
```

Returns the current file position for `stream`, or -1L on error.

fwrite

```
#include <stdio.h>
size_t fwrite( const void *ptr, size_t size, size_t nobj, FILE *stream );
```

Writes `nobj` members of `size` bytes to the given `stream` from the array pointed to by `ptr`.

Returns the number of successfully written objects.

getc

```
#include <stdio.h>
int getc( FILE *stream );
```

Reads one character out of the given `stream`. Currently #defined as getchar(), because FILE I/O is not supported. See also "_ioread()".

Returns the character read or EOF on error.

getchar

```
#include <stdio.h>
int getchar( void );
```

Reads one character from standard input. See also "_ioread()".

Returns the character read or EOF on error.

getenv

```
#include <stdlib.h>
char *getenv( const char *name );
```

Returns the environment string associated with `name`, or NULL if no string exists.

gets

```
#include <stdio.h>
char *gets( char *s );
```

Reads all characters from standard input until a newline is found. The newline is replaced by a NULL-character. See also "_ioread()".

Returns a pointer to the read string or NULL on error.

gmtime

```
#include <time.h>
struct tm *gmtime( const time_t *tp );
```

Converts the calender time `*tp` into Coordinated Universal Time (UTC).

Returns a structure representing the UTC, or NULL if UTC is not available.

isalnum

```
#include <ctype.h>
int isalnum( int c );
```

Returns a non-zero value when `c` is an alphabetic character or a number ([A–Z][a–z][0–9]).

isalpha

```
#include <ctype.h>
int isalpha( int c );
```

Returns a non-zero value when c is an alphabetic character ([A–Z][a–z]).

isascii

```
#include <ctype.h>
int isascii( int c );
```

Returns a non-zero value when c is in the range of 0 and 127. This is a non-ANSI function.

iscntrl

```
#include <ctype.h>
int iscntrl( int c );
```

Returns a non-zero value when c is a control character.

isdigit

```
#include <ctype.h>
int isdigit( int c );
```

Returns a non-zero value when c is a numeric character ([0–9]).

isgraph

```
#include <ctype.h>
int isgraph( int c );
```

Returns a non-zero value when c is printable, but not a space.

islower

```
#include <ctype.h>
int islower( int c );
```

Returns a non-zero value when c is a lowercase character ([a–z]).

isprint

```
#include <ctype.h>
int isprint( int c );
```

Returns a non-zero value when c is printable, including spaces.

ispunct

```
#include <ctype.h>
int ispunct( int c );
```

Returns a non-zero value when c is a punctuation character (such as '.', ',', '!', etc.).

isspace

```
#include <ctype.h>
int isspace( int c );
```

Returns a non-zero value when c is a space type character (space, tab, vertical tab, formfeed, linefeed, carriage return).

isupper

```
#include <ctype.h>
int isupper( int c );
```

Returns a non-zero value when `c` is an uppercase character ([A–Z]).

isxdigit

```
#include <ctype.h>
int isxdigit( int c );
```

Returns a non-zero value when `c` is a hexadecimal digit ([0–9][A–F][a–f]).

labs

```
#include <stdlib.h>
long labs( long n );
```

Returns the absolute value of the signed long argument.

ldiv

```
#include <stdlib.h>
ldiv_t ldiv( long num, long denom );
```

Both arguments are long integers. The returned quotient and remainder are also long integers.

Returns a structure containing the quotient and remainder of `num` divided by `denom`.

localeconv

```
#include <locale.h>
struct lconv *localeconv( void );
```

Sets the components of an object with type `struct lconv` with values appropriate for the formatting of numeric quantities according to the rules of the current locale.

Returns a pointer to the filled-in object.

localtime

```
#include <time.h>
struct tm *localtime( const time_t *tp );
```

Converts the calender time `*tp` into local time.

Returns a structure representing the local time.

longjmp

```
#include <setjmp.h>
void longjmp( jmp_buf env, int val );
```

Restores the environment previously saved with a call to setjmp(). The function calling the corresponding call to setjmp() may not be terminated yet. The value of `val` may not be zero.

malloc

```
#include <stdlib.h>
void *malloc( size_t size );
```

The allocated space is not initialized. The maximum space that can be allocated can be changed by customizing the heap size (see Section 1.3.5, "Heap"). By default no heap is allocated. When "malloc()" is used while no heap is defined, the locator gives an error. When "malloc()" is used within the small or compact data memory model, the heap from the locator description file must be moved from addressing mode 'data' to addressing mode 'data short'. Otherwise, locating the application results in locating errors.

Returns a pointer to space in external memory of `size` bytes length. NULL if there is not enough space left.

mblen

```
#include <stdlib.h>
int mblen( const char *s, size_t n );
```

Determines the number of bytes comprising the multi-byte character pointed to by `s`, if `s` is not a null pointer. Except that the shift state is not affected. At most `n` characters will be examined, starting at the character pointed to by `s`.

Returns the number of bytes, or 0 if `s` points to the NULL character, or -1 if the bytes do not form a valid multi-byte character.

mbstowcs

```
#include <stdlib.h>
size_t mbstowcs( wchar_t *pwcs, const char *s, size_t n );
```

Converts a sequence of multi-byte characters that begins in the initial shift state from the array pointed to by `s`, into a sequence of corresponding codes and stores these codes into the array pointed to by `pwcs`, stopping after `n` codes are stored or a code with value zero is stored.

Returns the number of array elements modified (not including a terminating zero code, if any), or (`size_t`)-1 if an invalid multi-byte character is encountered.

mbtowc

```
#include <stdlib.h>
int mbtowc( wchar_t *pwc, const char *s, size_t n );
```

Determines the number of bytes that comprise the multi-byte character pointed to by `s`. It then determines the code for value of type `wchar_t` that corresponds to that multi-byte character. If the multi-byte character is valid and `pwc` is not a NULL pointer, the mbtowc function stores the code in the object pointed to by `pwc`. At most `n` characters will be examined, starting at the character pointed to by `s`.

Returns the number of bytes, or 0 if `s` points to the NULL character, or -1 if the bytes do not form a valid multi-byte character.

memchr

```
#include <string.h>
void *memchr( const void *cs, int c, size_t n );
```

Checks the first `n` bytes of `cs` on the occurrence of character `c`.

Returns NULL when not found, otherwise a pointer to the found character is returned.

memcmp

```
#include <string.h>
int memcmp( const void *cs, const void *ct, size_t n );
```

Compares the first n bytes of cs with the contents of ct.

Returns a value < 0 if cs < ct,
0 if cs = = ct,
or a value > 0 if cs > ct.

memcpy

```
#include <string.h>
void *memcpy( void *s, const void *ct, size_t n );
```

Copies n characters from ct to s. No care is taken if the two objects overlap.

Returns s

memmove

```
#include <string.h>
void *memmove( void *s, const void *ct, size_t n );
```

Copies n characters from ct to s. Overlapping objects will be handled correctly.

Returns s

memset

```
#include <string.h>
void *memset( void *s, int c, size_t n );
```

Fills the first n bytes of s with character c.

Returns s

mktime

```
#include <time.h>
time_t mktime( struct tm *tp );
```

Converts the local time in the structure *tp into calendar time.

Returns the calendar time, or -1 if it cannot be represented.

offsetof

```
#include <stddef.h>
int offsetof( type, member );
```

Returns the offset for the given member in an object of type.

perror

```
#include <stdio.h>
void perror( const char *s );
```

Prints s and an implementation-defined error message corresponding to the integer errno, as if by:

```
fprintf( stderr, "%s: %s\n", s, "error message" );
```

The contents of the error message are the same as those returned by the strerror function with the argument errno. See also the "strerror()" function.

Returns nothing.

printf

```
#include <stdio.h>
int printf( const char *format,... );
```

Performs a formatted write to the standard output stream. See also "_iowrite()" and Section 5.2.3, "Printf and Scanf Formatting Routines".

Returns the number of characters written to the output stream.

The `format` string may contain plain text mixed with conversion specifiers. Each conversion specifier should be preceded by a '%' character. The conversion specifier should be build in order:

- Flags (in any order):
  - `-` specifies left adjustment of the converted argument.
  - `+` a number is always preceded with a sign character. + has higher precedence as space.
  - `space` a negative number is preceded with a sign, positive numbers with a space.
  - `0` specifies padding to the field width with zeros (only for numbers).
  - `#` specifies an alternate output form. For o, the first digit will be zero. For x or X, "0x" and "0X" will be prefixed to the number. For e, E, f, g, G, the output always contains a decimal point, trailing zeros are not removed.
- A number specifying a minimum field width. The converted argument is printed in a field with at least the length specified here. If the converted argument has fewer characters than specified, it will be padded at the left side (or at the right when the flag '-' was specified) with spaces. Padding to numeric fields will be done with zeros when the flag '`0`' is also specified (only when padding left). Instead of a numeric value, also '`*`' may be specified, the value is then taken from the next argument, which is assumed to be of type int.
- A period. This separates the minimum field width from the precision.
- A number specifying the maximum length of a string to be printed. Or the number of digits printed after the decimal point (only for floating point conversions). Or the minimum number of digits to be printed for an integer conversion. Instead of a numeric value, also '`*`' may be specified, the value is then taken from the next argument, which is assumed to be of type int.
- A length modifier 'h', 'l' or 'L'. 'h' indicates that the argument is to be treated as a short or unsigned short number. 'l' should be used if the argument is a long integer. 'L' indicates that the argument is a long double.

Flags, length specifier, period, precision and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined:

Character	Printed as
d, i	int, signed decimal
o	int, unsigned octal
x, X	int, unsigned hexadecimal in lowercase or uppercase respectively
u	int, unsigned decimal
c	int, single character (converted to unsigned char)
s	char *, the characters from the string are printed until a NULL character is found. When the given precision is met before, printing will also stop.
f	double
e, E	double
g, G	double
n	int *, the number of characters written so far is written into the argument. This should be a pointer to an integer in default memory. No value is printed.
p	pointer (hexadecimal 24-bit value)
%	No argument is converted, a '%' is printed.

putc

```
#include <stdio.h>
int putc( int c, FILE *stream );
```

Puts one character onto the given stream. See also "_iowrite()".

Returns EOF on error.

putchar

```
#include <stdio.h>
int putchar( int c );
```

Puts one character onto standard output. See also "_iowrite()".

Returns the character written or EOF on error.

puts

```
#include <stdio.h>
int puts( const char *s );
```

Writes the string to stdout, the string is terminated by a newline. See also "_iowrite()".

Returns NULL if successful, or EOF on error.

qsort

```
#include <stdlib.h>
void qsort( const void *base, size_t n, size_t size,
  int (* cmp)(const void *, const void *) );
```

This function sorts an array of `n` members. The initial base of the array is given by `base`. The size of each member is specified by `size`. The given array is sorted in ascending order, according to the results of the function pointed to by `cmp`.

raise

```
#include <signal.h>
int raise( int sig );
```

Sends the signal `sig` to the program. See also "signal()".

Returns zero if successful, or a non-zero value if unsuccessful.

rand

```
#include <stdlib.h>
int rand( void );
```

Returns a sequence of pseudo-random integers, in the range 0 to RAND_MAX.

realloc

```
#include <stdlib.h>
void *realloc( void *p, size_t size );
```

Reallocates the space for the object pointed to by `p`. The contents of the object will be the same as before calling realloc().The maximum space that can be allocated can be changed by customizing the heap size (see Section 1.3.5, "Heap"). By default no heap is allocated. When "realloc()" is used while no heap is defined, the linker gives an error. See also "malloc()".

Returns NULL and `*p` is not changed, if there is not enough space for the new allocation. Otherwise a pointer to the newly allocated space for the object is returned.

remove

```
#include <stdio.h>
int remove( const char *filename );
```

Removes the named file, so that a subsequent attempt to open it fails.

Returns zero if file is successfully removed, or a non-zero value, if the attempt fails.

rename

```
#include <stdio.h>
int rename( const char *oldname, const char *newname );
```

Changes the name of the file.

Returns zero if file is successfully renamed, or a non-zero value, if the attempt fails.

rewind

```
#include <stdio.h>
void rewind( FILE *stream );
```

Sets the file position indicator for the stream pointed to by `stream` to the beginning of the file. This function is equivalent to:

```
(void) fseek( stream, 0L, SEEK_SET );
clearerr( stream );
```

Returns nothing.

scanf

```
#include <stdio.h>
int scanf( const char *format, ... );
```

Performs a formatted read from the standard input stream. See also "_ioread()" and Section 1.5.2.3, "Printf and Scanf Formatting Routines".

Returns the number of items converted successfully.

All arguments to this function should be pointers to variables (in default memory) of the type which is specified in the format string.
The format string may contain:

- Blanks or tabs, which are skipped.
- Normal characters (not '%'), which should be matched exactly in the input stream.
- Conversion specifications, starting with a '%' character.

Conversion specifications should be build as follows (in order):

- A '*', meaning that no assignment is done for this field.
- A number specifying the maximum field width.
- The conversion characters `d`, `i`, `n`, `o`, `u` and `x` may be precede by 'h' if the argument is a pointer to `short` rather than `int`, or by 'l' (letter ell) if the argument is a pointer to `long`. The conversion characters `e`, `f`, and `g` may be precede by 'l' if a pointer `double` rather than `float` is in the argument list, and by 'L' if a pointer to a `long double`.
- A conversion specifier. '*', maximum field width and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined.

Length specifier and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined.

Character	Scanned as
d	int, signed decimal.
i	int, the integer may be given octal (i.e. a leading 0 is entered) or hexadecimal (leading "0x" or "0X"), or just decimal.
o	int, unsigned octal.
u	int, unsigned decimal.
x	int, unsigned hexadecimal in lowercase or uppercase.
c	single character (converted to unsigned char).
s	char *, a string of non white space characters. The argument should point to an array of characters, large enough to hold the string and a terminating NULL character.
f	float
e, E	float
g, G	float
n	int *, the number of characters written so far is written into the argument. No scanning is done.
p	pointer; hexadecimal 24-bit value.
[...]	Matches a string of input characters from the set between the brackets. A NULL character is added to terminate the string. Specifying [ ]...] includes the ']' character in the set of scanning characters.
[^...]	Matches a string of input characters not in the set between the brackets. A NULL character is added to terminate the string. Specifying [^]...] includes the ']' character in the set.
%	Literal '%', no assignment is done.

setbuf

```
#include <stdio.h>
void setbuf( FILE *stream, char *buf );
```

Buffering is turned off for the `stream`, if `buf` is NULL.

Otherwise, setbuf is equivalent to:

```
(void) setvbuf( stream, buf, _IOFBF, BUFSIZ )
```

See also "setvbuf( )".

setjmp

```
#include <setjmp.h>
int setjmp( jmp_buf env );
```

Saves the current environment for a subsequent call to longjmp.

Returns the value 0 after a direct call to setjmp(). Calling the function "longjmp()" using the saved `env` will restore the current environment and jump to this place with a non-zero return value.

See also "longjmp()".

setlocale

```
#include <locale.h>
char *setlocale( int category, const char *locale );
```

Selects the appropriate portion of the program's locale as specified by the `category` and `locale` arguments.

Returns the string associated with the specified `category` for the new locale if the selection can be honored.
NULL pointer if the selection cannot be honored.

setvbuf

```
#include <stdio.h>
int setvbuf( FILE *stream, char *buf, int mode, size_t size );
```

Controls buffering for the `stream`; this function must be called before reading or writing. `mode` can have the following values:

_IOFBF	causes full buffering
_IOLBF	causes line buffering of text files
_IONBF	causes no buffering

If `buf` is not NULL, it will be used as a buffer; otherwise a buffer will be allocated. `size` determines the buffer size.

Returns zero if successful, a non-zero value for an error.

See also "setbuf( )".

signal

```
#include <signal.h>
void (*signal( int sig, void (*handler)(int)))(int);
```

Determines how subsequent signals will be handled. If `handler` is SIG_DFL, the default behavior is used; if `handler` is SIG_IGN, the signal is ignored; otherwise, the function pointed to by `handler` will be called, with the argument of the type of signal. Valid signals are:

SIGABRT	abnormal termination, e.g. from `abort`
SIGFPE	arithmetic error, e.g. zero divide or overflow
SIGILL	illegal function image, e.g. illegal instruction
SIGINT	interactive attention, e.g. interrupt
SIGSEGV	illegal storage access, e.g. access outside memory limits
SIGTERM	termination request sent to this program

When a signal `sig` subsequently occurs, the signal is restored to its default behavior; then the signal-handler function is called, as if by `(*handler)(sig)`. If the handler returns, the execution will resume where it was when the signal occurred.

Returns the previous value of `handler` for the specific signal, or SIG_ERR if an error occurs.

sprintf

```
#include <stdio.h>
int sprintf( char *s, const char *format, ... );
```

Performs a formatted write to a string. See also "printf()" and Section 1.5.2.3, "Printf and Scanf Formatting Routines".

srand

```
#include <stdlib.h>
void srand( unsigned int seed );
```

This function uses `seed` as the start of a new sequence of pseudo-random numbers to be returned by subsequent calls to srand(). When srand is called with the same seed value, the sequence of pseudo-random numbers generated by rand() will be repeated.

Returns nothing.

sscanf

```
#include <stdio.h>
int sscanf( char *s, const char *format, ... );
```

Performs a formatted read from a string. See also "scanf()" and Section 1.5.2.3, "Printf and Scanf Formatting Routines".

strcat

```
#include <string.h>
char *strcat( char *s, const char *ct );
```

Concatenates string ct to string s, including the trailing NULL character.

Returns s

strchr

```
#include <string.h>
char *strchr( const char *cs, int c );
```

Returns a pointer to the first occurrence of character c in the string cs. If not found, NULL is returned.

strcmp

```
#include <string.h>
int strcmp( const char *cs, const char *ct );
```

Compares string cs to string ct.

Returns <0 if cs < ct,
0 if cs = = ct,
>0 if cs > ct.

strcoll

```
#include <string.h>
int strcoll( const char *cs, const char *ct );
```

Compares string cs to string ct. The comparison is based on strings interpreted as appropriate to the program's locale.

Returns <0 if cs < ct,
0 if cs = = ct,
>0 if cs > ct.

strcpy

```
#include <string.h>
char *strcpy( char *s, const char *ct );
```

Copies string ct into the string s, including the trailing NULL character.

Returns s

strcspn

```
#include <string.h>
size_t strcspn( const char *cs, const char *ct );
```

Returns the length of the prefix in string cs, consisting of characters not in the string ct.

strerror

```
#include <string.h>
char *strerror( size_t n );
```

Returns pointer to implementation-defined string corresponding to error n.

strftime

```
#include <time.h>
size_t
strftime( char *s, size_t smax, const char *fmt, const struct tm *tp );
```

Formats date and time information from the structure `*tp` into `s` according to the specified format `fmt`. `fmt` is analogous to a `printf` format. Each *%c* is replaced as described below:

%a	abbreviated weekday name
%A	full weekday name
%b	abbreviated month name
%B	full month name
%c	local date and time representation
%d	day of the month (01–31)
%H	hour, 24-hour clock (00–23)
%I	hour, 12-hour clock (01–12)
%j	day of the year (001–366)
%m	month (01–12)
%M	minute (00–59)
%p	local equivalent of AM or PM
%S	second (00–59)
%U	week number of the year, Sunday as first day of the week (00–53)
%w	weekday (0–6, Sunday is 0)
%W	week number of the year, Monday as first day of the week (00–53)
%x	local date representation
%X	local time representation
%y	year without century (00–99)
%Y	year with century
%Z	time zone name, if any
%%	%

Ordinary characters (including the terminating '\0') are copied into `s`. No more than `smax` characters are placed into `s`.

Returns the number of characters ('\0' not included), or
zero if more than `smax` characters where produced.

strlen

```
#include <string.h>
size_t strlen( const char *cs );
```

Returns the length of the string in `cs`, not counting the NULL character.

strncat

```
#include <string.h>
char *strncat( char *s, const char *ct, size_t n );
```

Concatenates string `ct` to string `s`, at most `n` characters are copied. Add a trailing NULL character.

Returns `s`

strncmp

```
#include <string.h>
int strncmp( const char *cs, const char *ct, size_t n );
```

Compares at most `n` bytes of string `cs` to string `ct`.

Returns <0 if `cs < ct`,
0 if `cs = = ct`,
>0 if `cs > ct`.

strncpy

```
#include <string.h>
char *strncpy( char *s, const char *ct, size_t n );
```

Copies string ct onto the string s, at most n characters are copied. Add a trailing NULL character if the string is smaller than n characters.

Returns s

strpbrk

```
#include <string.h>
char *strpbrk( const char *cs, const char *ct );
```

Returns a pointer to the first occurrence in cs of any character out of string ct. If none are found, NULL is returned.

strrchr

```
#include <string.h>
char *strrchr( const char *cs, int c );
```

Returns a pointer to the last occurrence of c in the string cs. If not found, NULL is returned.

strspn

```
#include <string.h>
size_t strspn( const char *cs, const char *ct );
```

Returns the length of the prefix in string cs, consisting of characters in the string ct.

strstr

```
#include <string.h>
char *strstr( const char *cs, const char *ct );
```

Returns a pointer to the first occurrence of string ct in the string cs. Returns NULL if not found.

strtod

```
#include <stdlib.h>
double strtod( const char *s, char **endp );
```

Converts the initial portion of the string pointed to by s to a double value. Initial white spaces are skipped. When endp is not a NULL pointer, after this function is called, *endp will point to the first character not used by the conversion.

Returns the read value.

strtok

```
#include <string.h>
char *strtok( char *s, const char *ct );
```

Search the string s for tokens delimited by characters from string ct. It terminates the token with a NULL character.

Returns a pointer to the token. A subsequent call with s == NULL will return the next token in the string.

strtol

```
#include <stdlib.h>
long strtol( const char *s, char **endp, int base );
```

Converts the initial portion of the string pointed to by `s` to a long integer. Initial white spaces are skipped. Then a value is read using the given `base`. When `base` is zero, the `base` is taken as defined for integer constants. I.e. numbers starting with an '0' are taken octal, numbers starting with '0x' or '0X' are taken hexadecimal. Other numbers are taken decimal. When `endp` is not a NULL pointer, after this function is called, `*endp` will point to the first character not used by the conversion.

Returns the read value.

strtoul

```
#include <stdlib.h>
unsigned long strtoul( const char *s, char **endp, int base );
```

Converts the initial portion of the string pointed to by `s` to an unsigned long integer. Initial white spaces are skipped. Then a value is read using the given `base`. When `base` is zero, the `base` is taken as defined for integer constants. I.e. numbers starting with an '0' are taken octal, numbers starting with '0x' or '0X' are taken hexadecimal. Other numbers are taken decimal. When `endp` is not a NULL pointer, after this function is called, `*endp` will point to the first character not used by the conversion.

Returns the read value.

strxfrm

```
#include <string.h>
size_t strncmp( char *ct, const char *cs, size_t n );
```

Transforms the string pointed to by `cs` and places the resulting string into the array pointed to by `ct`. No more than `n` characters are placed into the resulting string pointed to by `ct`, including the terminating null character.

Returns the length of the transformed string.

system

```
#include <stdlib.h>
int system( const char *s );
```

Passes the string `s` to the environment for execution.

Returns a non-zero value if there is a command processor, if `s` is NULL; or an implementation-dependent value, if `s` is not NULL.

time

```
#include <time.h>
time_t time( time_t *tp );
```

The return value is also assigned to `*tp`, if `tp` is not NULL.

Returns the current calendar time, or -1 if the time is not available.

tmpfile

```
#include <stdio.h>
FILE *tmpfile( void );
```

Creates a temporary file of the mode "`wb+`" that will be automatically removed when closed or when the program terminates normally.

Returns a stream if successful, or NULL if the file could not be created.

tmpnam

```
#include <stdio.h>
char *tmpnam( char s[L_tmpnam] );
```

Creates a temporary name (not a file). Each time `tmpnam` is called a different name is created. `tmpnam(NULL)` creates a string that is not the name of an existing file, and returns a pointer to an internal static array. `tmpnam(s)` creates a string and stores it in `s` and also returns it as the function value. `s` must have room for at least `L_tmpnam` characters. At most TMP_MAX different names are guaranteed during execution of the program.

Returns a pointer to the temporary name, as described above.

toascii

```
#include <ctype.h>
int toascii( int c );
```

Converts `c` to an ascii value (strip highest bit). This is a non-ANSI function.

Returns the converted value.

tolower

```
#include <ctype.h>
int tolower( int c );
```

Returns `c` converted to a lowercase character if it is an uppercase character, otherwise `c` is returned.

toupper

```
#include <ctype.h>
int toupper( int c );
```

Returns `c` converted to an uppercase character if it is a lowercase character, otherwise `c` is returned.

ungetc

```
#include <stdio.h>
int ungetc( int c, FILE *fin );
```

Pushes at the most one character back onto the input buffer.

Returns EOF on error.

va_arg

```
#include <stdarg.h>
va_arg( va_list ap, type );
```

Returns the value of the next argument in the variable argument list. It's return type has the type of the given argument `type`. A next call to this macro will return the value of the next argument.

va_end

```
#include <stdarg.h>
va_end( va_list ap );
```

This macro must be called after the arguments have been processed. It should be called before the function using the macro 'va_start' is terminated (ANSI specification).

va_start

```
#include <stdarg.h>
va_start( va_list ap, lastarg );
```

This macro initializes `ap`. After this call, each call to va_arg() will return the value of the next argument. In our implementation, `va_list` cannot contain any bit type variables. Also the given argument `lastarg` must be the last non-bit type argument in the list.

vfprintf

```
#include <stdio.h>
int vfprintf( FILE *stream, const char *format, va_list arg );
```

Is equivalent to vprintf, but writes to the given stream. See also "vprintf()", "_iowrite()" and Section 1.5.2.3, "Printf and Scanf Formatting Routines".

vprintf

```
#include <stdio.h>
int vprintf( const char *format, va_list arg );
```

Does a formatted write to standard output. Instead of a variable argument list as for printf(), this function expects a pointer to the list. See also "printf()", "_iowrite()" and Section 1.5.2.3, "Printf and Scanf Formatting Routines".

vsprintf

```
#include <stdio.h>
int vsprintf( char *s, const char *format, va_list arg );
```

Does a formatted write a string. Instead of a variable argument list as for printf(), this function expects a pointer to the list. See also "printf()", "_iowrite()" and Section 1.5.2.3, "Printf and Scanf Formatting Routines".

wcstombs

```
#include <stdlib.h>
size_t wcstombs( char *s, const wchar_t *pwcs, size_t n );
```

Converts a sequence of codes that correspond to multi-byte characters from the array pointed to by `pwcs`, into a sequence of multi-byte characters that begins in the initial shift state and stores these multi-byte characters into the array pointed to by `s`, stopping if a multi-byte character would exceed the limit of `n` total bytes or if a NULL character is stored.

Returns the number of bytes modified (not including a terminating NULL character, if any), or (`size_t`)-1 if a code is encountered that does not correspond to a valid multi-byte character.

wctomb

```
#include <stdlib.h>
int wctomb( char *s, wchar_t wchar );
```

Determines the number of bytes needed to represent the multi-byte corresponding to the code whose value is `wchar` (including any change in the shift state). It stores the multi-byte character representation in the array pointed to by `s` (if `s` is not a NULL pointer). At most MB_CUR_MAX characters are stored. If the value of `wchar` is zero, the wctomb function is left in the initial shift state.

Returns the number of bytes, or -1 if the value of `wchar` does not correspond to a valid multi-byte character.

1.5.2.3 Printf and Scanf Formatting Routines

The functions `printf()`, `fprintf()`, `vfprintf()`, `vsprintf()`, ... call one single function that deals with the format string and arguments. This function is `_doprint()`. This is a rather big function because the number of possibilities of the format specifiers in a format string are large. If you do not use all the possibilities of the format specifiers a smaller `_doprint()` function can be used. Three different versions exist:

LARGE	the full formatter, no restrictions
MEDIUM	floating point printing is not supported
SMALL	as MEDIUM, but also the precision specifier '.' cannot be used

The same applies to all `scanf` type functions, which all call the function `_doscan()`.

The formatters included in the libraries are LARGE. You can select different formatters by linking separate objects of `_doscan()` and `_doprint()` with your application. The following objects are included:

`lib\libcs`

`_doprnts.obj`	`_doprint()`, small model, SMALL formatter
`_doprntm.obj`	`_doprint()`, small model, MEDIUM formatter
`_doprntl.obj`	`_doprint()`, small model, LARGE formatter
`_doscans.obj`	`_doscan()`, small model, SMALL formatter
`_doscanm.obj`	`_doscan()`, small model, MEDIUM formatter
`_doscanl.obj`	`_doscan()`, small model, LARGE formatter

`lib\libcc`

`_doprnts.obj`	`_doprint()`, code compact, SMALL formatter
`_doprntm.obj`	`_doprint()`, code compact, MEDIUM formatter
`_doprntl.obj`	`_doprint()`, code compact, LARGE formatter
`_doscans.obj`	`_doscan()`, code compact, SMALL formatter
`_doscanm.obj`	`_doscan()`, code compact, MEDIUM formatter
`_doscanl.obj`	`_doscan()`, code compact, LARGE formatter

`lib\libcd`

`_doprnts.obj`	`_doprint()`, data compact, SMALL formatter
`_doprntm.obj`	`_doprint()`, data compact, MEDIUM formatter
`_doprntl.obj`	`_doprint()`, data compact, LARGE formatter
`_doscans.obj`	`_doscan()`, data compact, SMALL formatter
`_doscanm.obj`	`_doscan()`, data compact, MEDIUM formatter
`_doscanl.obj`	`_doscan()`, data compact, LARGE formatter

`lib\libcl`

`_doprnts.obj`	`_doprint()`, large model, SMALL formatter
`_doprntm.obj`	`_doprint()`, large model, MEDIUM formatter
`_doprntl.obj`	`_doprint()`, large model, LARGE formatter
`_doscans.obj`	`_doscan()`, large model, SMALL formatter
`_doscanm.obj`	`_doscan()`, large model, MEDIUM formatter
`_doscanl.obj`	`_doscan()`, large model, LARGE formatter

Example:

```
cc88 -Ms hello.obj c:\c88\lib\libcs\_doprntm.obj
```

This will use the MEDIUM `_doprint()` formatter for the small model.

1.5.3 Run-time Library

Some compiler generated code contains calls to run-time library functions that would use too much code if generated as inline code. The name of a run-time library function always contains two leading underscores. For example, to perform a 32 bit division.

Because **c88** generates assembly code (and not object code) it prepends an underscore '_' for the names of (public) C variables to distinguish these symbols from S1C88 registers. So if you use a function with a leading underscore, the assembly label for this function contains two leading underscores. This function name could cause a name conflict (double defined) with one of the run-time library functions. However, ANSI states that it is not portable to use names starting with an underscore for public C variables and functions, because results are implementation defined.

Table 1.5.3.1 Run-time library name syntax

Compiler Model	Library to link
small (default)	librts.a (default)
compact code	librtc.a
compact data	librtd.a
large	librtl.a

1.6 Floating Point Arithmetic

Floating point arithmetic support for the **c88** is included in software as a separate set of libraries. When linking, the desired floating point library must be specified after the C library. The libraries are reentrant, and only use temporary program stack memory.

To ensure portability of floating point arithmetic, floating point arithmetic for the **c88** has been implemented adhering to the IEEE-754 standard for floating point arithmetic. See the "IEEE Standard for Binary Floating-Point Arithmetic" document, as published in 1985 by the IEEE Computer Society, for more details on these floating point arithmetic definitions. This document is referred to as IEEE-754 in this manual.

c88 supports single precision floating point operations only, usable via the ANSI C types `float` and `double`. For the sole purpose of speed, also a non-trapping library is included for each memory model. For the library name syntax, see Section 1.6.6, "Floating Point Libraries".

It is possible to intercept floating point exceptional cases and, if desired, handle them with an application defined exception handler. The intercepting of floating point exceptions is referred to as 'trapping'. Examples of how to install a trap handler are included.

1.6.1 Data Size and Register Usage

c88 handles `float` and `double` type values as 4-bit data. The range that can be specified is

+/-1,176E-38 to +/-3,402E+38

The compiler uses the HLBA register (HL ← high word, BA ← low word) when `float/double` type arguments and return values are allocated to a register.

1.6.2 Compiler Option

-F and **-Fc** shown below are provided as the command option for **c88** to controlling floating point arithmetic. See Section 1.4.2, "Compiler", for details of the invocation syntax and other options.

-F/-Fc

Option:

-F[**c**]

Description:

-F forces using single precision floating point only, even when `double` or `long double` is used. In fact `double` and `long double` are treated as `float` and default argument promotion from `float` to `double` is suppressed. When you use this option, you must use the single precision version of the C library. See Section 1.6.6, "Floating Point Libraries", for the naming conventions of the standard libraries.

-Fc enables the use of 'float' constants. In ANSI C floating point constants are treated having type `double`, unless the constant has the suffix '**f**'. So '3.0' is a double precision constant, while '3.0f' is a single precision constant. This option tells the compiler to treat all floating point constants as single precision float types (unless they have an explicit 'l' suffix).

Example:

To force `double` to be treated as `float`, enter:

```
c88 -F test.c
```

1.6.3 Special Floating Point Values

Below is a list of special, IEEE-754 defined, floating point values as they can occur during run-time.

Table 1.6.3.1 Special floating point values

Special Value	Sign	Exponent	Mantissa
+0.0 (Positive Zero)	0	all zeros	all zeros
-0.0 (Negative Zero)	1	all zeros	all zeros
+INF (Positive Infinite)	0	all ones	all zeros
-INF (Negative Infinite)	1	all zeros	all zeros
NaN (Not a Number)	0	all ones	all ones

1.6.4 Trapping Floating Point Exceptions

Two floating point run-time libraries are delivered for every memory model:

with floating point trap handling (`libfp`*`m`*`t.a`)

without a trapping mechanism (`libfp`*`m`*`.a`)

The distinction is made by adding an additional 't' to the name of the library comprising trap handling. The *m* must be replaced by one of the C memory models ('s' small, 'c' compact code, 'd' compact data or 'l' large). By specifying the **-fptrap** option to the control program **cc88**, the trapping type floating point library is linked into your application. If this option is not specified, the floating point library without any trapping mechanism is used when linking.

The floating point libraries without trapping mechanism execute faster, but the result of a floating point operation is undefined when any operand or result is not in range.

IEEE-754 Trap Handler

In the IEEE-754 standard a trap handler is defined, which is invoked on (specified) exceptional events, passing along much information about the event. To install your own trap handler, use the library call `_fp_install_trap_handler`. When installing your own exception handler, you will have to select on which types of exceptions you want to have your handler invoked, using the function call `_fp_set_exception_mask`. See further below for more details on the floating point library exception handling function interface.

SIGFPE Signal Handler

In ANSI-C the regular approach of dealing with floating point exceptions is by installing a so-called signal handler by means of the ANSI-C library call `signal`. If such a handler is installed, floating point exceptions cause this handler to be invoked. To have the signal handler for the **SIGFPE** signal actually become operational with the provided floating point libraries, a (very) basic version of the IEEE-754 exception handler must be installed (see example below) which will raise the desired signal by means of the ANSI-C library function call `raise`. For this to be achieved, the function call `_fp_install_trap _handler` is present. When installing your own exception handler, you will have to select on which types of exceptions you want to receive a signal, using the function call `_fp_set_exception_mask`. See further below for more details on the floating point library exception handling function interface.

There is no way to specify any information about the context or nature of the exception to the signal handler. Just that a floating point exception occurred can be detected. See therefore the IEEE-754 trap handler discussion above if you want more control over floating point results.

Example:

```
#include <float.h>
#include <signal.h>

static void pass_fp_exception_to_signal( _fp_exception_info_t *info )
{
        info;   /* suppress parameter not used warning */

        /* cause SIGFPE signal to be raised */

        raise( SIGFPE );
        /*
         * now continue the program
         * with the unaltered result
         */
}
```

1.6.5 Floating Point Trap Handling API

For purposes of dealing with floating point arithmetic exceptions, the following library calls are available:

```
#include <float.h>

int     _fp_get_exception_mask( void );
void    _fp_set_exception_mask( int );
```

A pair of functions to get or set the mask which controls which type of floating point arithmetic exceptions are either ignored or passed on to the trap handler. The types of possible exception flag bits are defined as:

EFINVOP
EFDIVZ
EFOVFL
EFUNFL
EFINEXCT

while,

EFALL

is the OR of all possible flags. See below for an explanation of each flag.

```
#include <float.h>

int     _fp_get_exception_status( void );
void    _fp_set_exception_status( int );
```

A pair of functions for examining or presetting the status word containing the accumulation of all floating point exception types which occurred so far. See the possible exception type flags above.

```
#include <float.h>

void    _fp_install_trap_handler( void (*)(_fp_exception_info_t * ) );
```

This function call expects a pointer to a function, which in turn expects a pointer to a structure of type `_fp_exception_info_t`. The members of `_fp_exception_info_t` are:

exception

This member contains one of the following (numerical) values:

EFINVOP
EFDIVZ
EFOVFL
EFUNFL
EFINEXCT

operation

This member contains one of the following numbers:

_OP_ADDITION
_OP_SUBTRACTION
_OP_COMPARISON
_OP_EQUALITY
_OP_LESS_THAN
_OP_LARGER_THAN
_OP_MULTIPLICATION
_OP_DIVISION
_OP_CONVERSION

source_format
destination_format

Numerical values of these two members are:

_TYPE_SIGNED_CHARACTER
_TYPE_UNSIGNED_CHARACTER
_TYPE_SIGNED_SHORT_INTEGER
_TYPE_UNSIGNED_SHORT_INTEGER
_TYPE_SIGNED_INTEGER
_TYPE_UNSIGNED_INTEGER
_TYPE_SIGNED_LONG_INTEGER
_TYPE_UNSIGNED_LONG_INTEGER
_TYPE_FLOAT
_TYPE_DOUBLE

```
operand1        /* left side of binary or */
                /* right side of unary */
operand2        /* right side for binary */
result
```

These three are of the following type, to receive and return a value of arbitrary type:

```
typedef union _fp_value_union_t
{
        char            c;
        unsigned char   uc;
        short           s;
        unsigned short  us;
        int             i;
        unsigned int    ui;
        long            l;
        unsigned long   ul;
        float           f;
#if ! _SINGLE_FP
        double          d;
#endif
}
_fp_value_union_t;
```

The following table lists all the exception code flags, the corresponding error description and result:

Table 1.6.5.1 Exception type flag codes

Error Description	Exception Flag	Default Result with Trapping
Invalid Operation	EFINVOP	NaN
Division by zero	EFDIVZ	+INF or -INF
Overflow	EFOVFL	+INF or -INF
Underflow	EFUNFL	zero
Inexact	EFINEXT	undefined
INF Infinite which is the largest absolute floating point number, being always: -INF < every finite number < +INF		
NAN Not a Number, a symbolic entity encoded in floating point format.		

To ensure all exception types are specified, you can specify **EFALL** to a function, which is the binary OR of all above enlisted flags.

1.6.6 Floating Point Libraries

When you use floating point, the floating point library must always be linked after the C library and before the run-time library. Arithmetic routines like `sin()`, `cos()`, etc. are not present in these libraries (they are present in the C library), only basic floating point operations can be done.

Table 1.6.6.1 Compiler model and floating point library

Compiler Model	Library to link	
	No trapping	Trapping
Small (default)	libfps.a (default)	libfpst.a
Compact code	libfpc.a	libfpct.a
Compact data	libfpd.a	libfpdt.a
Large	libfpl.a	libfplt.a

The following floating point header files are delivered with the C compiler:

<float.h> Constants related to floating point arithmetic.

<math.h> acos, asin, atan, atan2, ceil, cos, cosh, exp, fabs, floor, fmod, frexp, ldexp, log, log10, modf, pow, sin, sinh, sqrt, tan, tanh

<time.h> difftime (Delivered as a skeleton.)

1.6.6.1 Floating Point Arithmetic Routine

acos

```
#include <math.h>
double acos( double x );
```

Returns the arccosine $\cos^{-1}(x)$ of x in the range $[0, \pi]$, $x \in [-1, 1]$.

asin

```
#include <math.h>
double asin( double x );
```

Returns the arcsine $\sin^{-1}(x)$ of x in the range $[-\pi/2, \pi/2]$, $x \in [-1, 1]$.

atan

```
#include <math.h>
double atan( double x );
```

Returns the arctangent $\tan^{-1}(x)$ of x in the range $[-\pi/2, \pi/2]$, $x \in [-1, 1]$.

atan2

```
#include <math.h>
double atan2( double y, double x );
```

Returns the result of: $\tan^{-1}(y/x)$ in the range $[-\pi, \pi]$.

atof

```
#include <stdlib.h>
double atof( const char *s );
```

Converts the given string to a double value. White space is skipped, conversion is terminated at the first unrecognized character.

Returns the double value.

ceil

```
#include <math.h>
double ceil( double x );
```

Returns the smallest integer not less than x, as a double.

cos

```
#include <math.h>
double cos( double x );
```

Returns the cosine of x.

cosh

```
#include <math.h>
double cosh( double x );
```

Returns the hyperbolic cosine of x.

difftime

```
#include <time.h>
double difftime( time_t time2, time_t time1 );
```

Returns the result of `time2 - time1` in seconds.

exp

```
#include <math.h>
double exp( double x );
```

Returns the result of the exponential function e^x.

fabs

```
#include <math.h>
double fabs( double x );
```

Returns the absolute double value of x. |x|

floor

```
#include <math.h>
double floor( double x );
```

Returns the largest integer not greater than x, as a double.

fmod

```
#include <math.h>
double fmod( double x, double y );
```

Returns the floating-point remainder of x/y, with the same sign as x. If y is zero, the result is implementation-defined.

frexp

```
#include <math.h>
double frexp( double x, int *exp );
```

Splits x into a normalized fraction in the interval [1/2, 1> //C-51 compatible, which is returned, and a power of 2, which is stored in `*exp`. If x is zero, both parts of the result are zero. For example: `frexp( 4.0, &var )` results in $0.5 \cdot 2^3$. The function returns 0.5, and 3 is stored in var.

Returns the normalized fraction.

ldexp

```
#include <math.h>
double ldexp( double x, int n );
```

Returns the result of: $x \cdot 2^n$.

log

```
#include <math.h>
double log( double x );
```

Returns the natural logarithm `ln(x)`, `x>0`.

log10

```
#include <math.h>
double log10( double x );
```

Returns the base 10 logarithm `log10(x)`, `x>0`.

modf

```
#include <math.h>
double modf( double x, double *ip );
```

Splits `x` into integral and fractional parts, each with the same sign as `x`. It stores the integral part in `*ip`.

Returns the fractional part.

pow

```
#include <math.h>
double pow( double x, double y );
```

A domain error occurs if `x=0` and `y 0`, or if `x<0` and `y` is not an integer.

Returns the result of `x` raised to the power of `y`: x^y.

sin

```
#include <math.h>
double sin( double x );
```

Returns the sine of `x`.

sinh

```
#include <math.h>
double sinh( double x );
```

Returns the hyperbolic sine of `x`.

sqrt

```
#include <math.h>
double sqrt( double x );
```

Returns the square root of `x`. $\sqrt{x}$, where `x` 0.

tan

```
#include <math.h>
double tan( double x );
```

Returns the tangent of `x`.

tanh

```
#include <math.h>
double tanh( double x );
```

Returns the hyperbolic tangent of `x`.

CHAPTER 2 ASSEMBLER

2.1 Description

The S1C88 assembler **as88** assembles the assembly source files generated by the C compiler **c88** to generate the relocatable object files that can be linked using **lk88**.

The following phases can be identified during assembly:

1. Preprocess
2. Legality check of all instructions
3. Address calculation
4. Generation of object and (when requested) list file

The assembler generates relocatable object files using the IEEE-695 object format. This file format specifies a code part and a symbol part as well as a symbolic debug information part.

File inclusion and macro facilities are integrated into the assembler. See Section 2.5, "Macro Operations", for more information.

2.1.1 Invocation

The compiler control program, **cc88**, may call the assembler automatically. **cc88** translates some of its command line options to options of **as88**. However, the assembler can be invoked as an individual program also.

The invocation of **as88** is:

```
as88    [option]... source-file [map-file]
as88    -V
```

Invocation with **-V** only displays a version header.

The *source-file* must be an assembly source file. This file is the input source of the assembler. This file contains assembly code which is either user written or generated by **c88**. Any name is allowed for this file. If this name does not have an extension, the extension `.asm` is assumed or, if the file is still not found, the extension `.src` is assumed.

The optional *map-file* is passed to the assembler when producing an absolute list file. The map file is produced by the locator. To produce an absolute list file, see Section 2.1.4.1, "Absolute List File Generation".

In the default situation, an object file with extension `.obj` is produced. With the **-l** option a list file with extension `.lst` is produced.

Options are preceded by a '-' (minus sign). Options can not be combined after a single '-'. If all goes well, the assembler generates a relocatable object module which contains the object code, with the default extension `.obj`. You can specify another output filename with the **-o** option. Error messages are written to the terminal, unless they are directed to an error list file with the **-err** assembler option.

The following list describes the assembler options briefly. The next section gives a more detailed description.

Options Summary

Option	Description
-C *file*	Include *file* before source
-D*macro*[=*def*]	Define preprocessor *macro*
-L[*flag...*]	Remove specified source lines from list file
-M[**s**|**c**|**d**|**l**]	Specify memory model
-V	Display version header only
-c	Switch to case insensitive mode (default case sensitive)
-e	Remove object file on assembly errors
-err	Redirect error messages to error file
-f *file*	Read options from *file*
-i[**l**|**g**]	Default label style local or global
-l	Generate listing file
-o *filename*	Specify name of output file
-t	Display section summary
-v	Verbose mode. Print the filenames and numbers of the passes while they progress
-w[*num*]	Suppress one or all warning messages

2.1.2 Detailed Description of Assembler Options

-C

Option:

-C *file*

Arguments:

The name of an include file.

Description:

Include *file* before assembling the source.

Example:

To include the file `S1C88.inc` before any other include file, enter:

```
as88 -C S1C88.inc test.src
```

-c

Option:

-c

Default:

Case sensitive

Description:

Switch to case insensitive mode. By default, the assembler operates in case sensitive mode.

Example:

To switch to case insensitive mode, enter:

```
as88 -c test.src
```

-D

Option:

-D*macro*[=*def*]

Arguments:

The macro you want to define and optionally its definition.

Description:

Define *macro* as in 'define'. If *def* is not given ('=' is absent), '1' is assumed. Any number of symbols can be defined.

Example:

```
as88 -DTWO=2 test.src
```

-e

Option:

-e

Description:

Use this option if you do not want an object file when the assembler generates errors. With this option the 'make' utility always does the proper productions.

Example:

```
as88 -e test.src
```

-err

Option:

-err

Description:

The assembler redirects error messages to a file with the same basename as the output file and the extension `.ers`. The assembler uses the basename of the output file instead of the input file.

Example:

To write errors to the `test.ers` instead of stderr, enter:

```
as88 -err test.src
```

-f

Option:

-f *file*

Arguments:

A filename for command line processing. The filename "-" may be used to denote standard input.

Description:

Use *file* for command line processing. To get around the limits on the size of the command line, it is possible to use command files. These command files contain the options that could not be part of the real command line. Command files can also be generated on the fly, for example by the make utility.

More than one **-f** option is allowed.

Some simple rules apply to the format of the command file:

1. It is possible to have multiple arguments on the same line in the command file.
2. To include whitespace in the argument, surround the argument with either single or double quotes.
3. If single or double quotes are to be used inside a quoted argument, we have to go by the following rules:
   a. If the embedded quotes are only single or double quotes, use the opposite quote around the argument. Thus, if a argument should contain a double quote, surround the argument with single quotes.
   b. If both types of quotes are used, we have to split the argument in such a way that each embedded quote is surrounded by the opposite type of quote.

Example:
```
"This has a single quote ' embedded"
```
or
```
'This has a double quote " embedded'
```
or
```
'This has a double quote " and a single quote '"' embedded"
```

4. Some operating systems impose limits on the length of lines within a text file. To circumvent this limitation it is possible to use continuation lines. These lines end with a backslash and newline. In a quoted argument, continuation lines will be appended without stripping any whitespace on the next line. For non-quoted arguments, all whitespace on the next line will be stripped.

   Example:
```
"This is a continuation \
line"
        → "This is a continuation line"

control(file1(mode,type),\
        file2(type))
        → control(file1(mode,type),file2(type))
```

5. It is possible to nest command line files up to 25 levels.

Example:

Suppose the file `mycmds` contains the following line:

```
-err
test.src
```

The command line can now be:

```
as88 -f mycmds
```

-i

Option:

-i[**l** | **g**]

Default:

-il (local labels)

Description:

Select default handling for label identifiers. **-il** specifies that data and code assembly labels are by default treated as LOCAL labels, unless overruled by the GLOBAL directive. With **-ig** data and code assembly labels are by default treated as GLOBAL labels, unless overruled by the LOCAL directive.

Example:

To specify that assembly label identifiers are treated as GLOBAL labels by default, enter:

```
as88 -ig test.src
```

-L

Option:

-**L**[*flag*...]

Arguments:

Optionally one or more flags specifying which source lines are to be removed from the list file.

Default:

-LcDEGlMnPQsWXy

Description:

Specify which source lines are to be removed from the list file. A list file is generated when the **-l** option is specified. If you do not specify the **-L** option the assembler removes source lines containing #line directives or symbolic debug information, empty source lines and puts wrapped source lines on one line. **-L** without any flags, is equivalent to **-Lcdeglmnpqswxy**, which removes all specified source lines form the list file.

Flags can be switched on with the lower case letter and switched off with the uppercase letter. The following flags are allowed:

c Default. Remove source lines containing assembler controls (the OPTIMIZE directive).

C Keep source lines containing assembler controls.

d Remove source lines containing section directives (the DEFSECT, SECT directives).

D Default. Keep source lines containing section directives.

e Remove source lines containing one of the symbol definition directives EXTERN, GLOBAL or LOCAL.

E Default. Keep source lines containing symbol definition directives.

g Remove generic instruction expansion.

G Default. Show generic instruction expansion.

l Default. Remove source lines containing C preprocessor line information (lines with #line).

L Keep source lines containing C preprocessor line information.

m Remove source lines containing macro/dup directives (lines with MACRO or DUP).

M Default. Keep source lines containing macro/dup directives.

n Default. Remove empty source lines (newlines).

N Keep empty source lines.

p Remove source lines containing conditional assembly (lines with IF, ELSE, ENDIF). Only the valid condition is shown.

P Default. Keep source lines containing conditional assembly.

q Remove source lines containing assembler equates (lines with EQU).

Q Default. Keep source lines containing assembler equates.

s Default. Remove source lines containing high level language symbolic debug information (lines with SYMB).

S Keep source lines containing HLL symbolic debug information.

w Remove wrapped part of source lines.

W Default. Keep wrapped source lines.

x Remove source lines containing MACRO/DUP expansions.

X Default. Keep source lines containing MACRO/DUP expansions.

y Default. Hide cycle counts.

Y Show cycle counts.

Example:

To remove source lines with assembler controls from the resulting list file and to remove wrapped source lines, enter:

```
as88 -l -Lcw test.src
```

-l

Option:

-l

Description:

Generate listing file. The listing file has the same basename as the output file. The extension is `.lst`.

Example:

To generate a list file with the name `test.lst`, enter:

```
as88 -l test.src
```

See also:

-L

-M

Option:

-M*model*

Arguments:

The memory model to be used, where *model* is one of:

s	small,	maximum of 64K code and data
c	compact code,	maximum of 64K code and 16M data
d	compact data,	maximum of 8M code and 64K data
l	large,	maximum of 8M code and 16M data

Default:

-Ml

Description:

Specify the memory model to be used for assembling source files.

Example:

To assemble using the small model, enter:

```
as88 -Ms test.src
```

-o

Option:

-o *filename*

Arguments:

An output filename. The filename may not start immediately after the option. There must be a tab or space in between.

Default:

Basename of assembly file with `.obj` suffix.

Description:

Use *filename* as output filename of the assembler, instead of the basename of the assembly file with the `.obj` extension.

Example:

To create the object file `myfile.obj` instead of `test.obj`, enter:

```
as88 test.src -o myfile.obj
```

-t

Option:

-t

Description:

Produce totals (section size summary). For each section its memory address, size, number of cycles and name is listed on `stdout`.

Example:

```
as88 -t test.src
```

```
Section summary:
 NR ADDR   SIZE CYCLE NAME
  1        0007     5 .text
  2 021234 000e     0 .data
  3        0001     0 .tiny
```

-V

Option:

-V

Description:

With this option you can display the version header of the assembler. This option must be the only argument of **as88**. Other options are ignored. The assembler exits after displaying the version header.

Example:

```
as88 -V
```

```
S1C88 assembler va.b rc       SN000000-015 (c) year TASKING, Inc.
```

-v

Option:

-v

Description:

Verbose mode. With this option specified, the assembler prints the filenames and the assembly passes while they progress. So you can see the current status of the assembler.

Example:

```
as88 -v test.src
```

```
Parsing "test.src"
   30 lines (total now 31)
Optimizing
Evaluating absolute ORG addresses
Parsing symbolic debug information
Creating object file "test.obj"
Closing object file
```

-W

Option:

-w[*num*]

Arguments:

Optionally the warning number to suppress.

Description:

-w suppress all warning messages. **-w***num* suppresses warning messages with number *num*. More than one **-w***num* option is allowed.

Example:

The following example suppresses warnings 113 and 114:

```
as88 -w113 -w114 file.src
```

2.1.3 Environment Variables used by as88

AS88INC — With this environment variable you can specify directories where the **as88** assembler will search for include files. Multiple pathnames can be separated with semicolons.

Include files whose names are enclosed in "" are searched for first in the directory of the file containing the include line, then in the current directory. If the include file is still not found, the assembler searches in a directory specified with this environment variable AS88INC. AS88INC contain more than one directory. Finally, the directory `..\include` relative to the directory where the assembler binary is located is searched.

For include files whose names are in <>, the directory of the file containing the include line and the current directory are not searched. However, the directories specified with AS88INC and the relative path are still searched.

TMPDIR — With the TMPDIR environment symbol you can specify the directory where the assembler can generate temporary files. If the assembler terminates normally, the temporary file will be removed automatically.
If you do not set TMPDIR, the temporary file will be created in the current working directory.

2.1.4 List File

The list file is the output file of the assembler which contains information about the generated code. The amount and form of information depends on the use of the **-L** option. The name is the basename of the output file with the extension `.lst`. The list file is only generated when the **-l** option is supplied. When **-l** is supplied, a list file is also generated when assembly errors/warnings occur. In this case the error/warning is given just below the source line containing the error/warning.

2.1.4.1 Absolute List File Generation

After locating the whole application, an absolute list file can be generated for all assembly source input files with the assembler. To generate an absolute list file from an assembly source file the source code needs to be assembled again with use of the locator map file of the application the assembly source belongs to. See Section 4.5, "Locator Output", how to produce a locator map file.

An absolute list file contains absolute addresses whereas a standard list file contains relocatable addresses.

When a map file is specified as input for the assembler, only the absolute list file is generated when list file generation is enabled with the list file option **-l**. The previously generated object file is not overwritten when absolute list file generation is enabled. Absolute list file generation is only enabled when a map file is specified on the input which contains the filename extension `.map`.

Note: When you want to generate an absolute list file, you have to specify the same options as you did when generating the object file. If the options are not the same you might get an incorrect absolute list file.

Example:

Suppose your first invocation was:

```
as88 -Ms test.src
```

then when you want to generate an absolute list file you have to specify the same option (**-Ms**) and the **-l** option:

```
as88 -Ms -l test.src test.map
```

With this command he absolute list file "test.lst" is created.

2.1.4.2 Page Header

The page header consists of four lines.

The first line contains the following information:

- information about assembler name
- version and serial number
- copyright notice

The second line contains a title specified by the TITLE (first page) or STITLE (succeeding pages) control and a page number.

The third line contains the name of the file (first page) or is empty (succeeding pages).

The fourth line contains the header of the source listing as described in the next section.

Example:

```
S1C88 assembler va.b rc        SNzzzzzz-zzz (c) year TASKING, Inc.
Title for demo use only                                    page   1
/tmp/hello.asm
ADDR   CODE      CYCLES LINE SOURCELINE
```

2.1.4.3 Source Listing

The following line appears in the page header:

```
ADDR   CODE      CYCLES LINE SOURCELINE
```

The different columns are discussed below.

ADDR

This is the memory address. The address is a (6 digit) hexadecimal number that represents the offset from the beginning of a relocatable section or the absolute address for an absolute section.

In lines that generate object code, the value is at the beginning of the line. For any other line there is no display.

Example:

```
ADDR   CODE      LINE SOURCELINE
000000              1         defsect ".text", code
000000              2         sect    ".text"
000000 CEC6rr       4         ld      xp,#@dpag(data_label)
000003 CEC4rr       7         ld      nb,#@cpag(label)
000006 F101         8         jr      label
                    .
                    .
021234             13         defsect ".data", data at 21234h
                   14 data_label:
021234             16         ds      49
   |  RESERVED
021264
```

CODE

This is the object code generated by the assembler for this source line, displayed in hexadecimal format. The displayed code need not be the same as the generated code that is entered in the object module. The code can also be relocatable code or a relocatable part and external part. In this case the letter 'r' is printed for the relocatable code part in the listing.

For lines that allocate space (DS) the code field contains the text "RESERVED".

Example:

```
ADDR   CODE      LINE SOURCELINE
                    .
                    .
000000 CEC6rr       4         ld      xp,#@dpag(data_label)
000003 CEC4rr       7         ld      nb,#@cpag(label)
000006 F101         8         jr      label
                    .
                    .
021234             13         defsect ".data", data at 21234h
                   14 data_label:
021234             16         ds      49
   |  RESERVED
021264
```

In this example the word "RESERVED" marks the space reserved for the **ds** directive.

CYCLES

If you provide the option **-LY** to the assembler, the CYCLES column also appears in the list file. The first value indicates the cycle count of the instruction, the second value is a cumulated cycle count.

Example:

```
ADDR   CODE      CYCLES LINE SOURCELINE
                           .
                           .
000000 CEC6rr    3    3    4         ld      xp,#@dpag(data_label)
000003 CEC4rr    4    7    7         ld      nb,#@cpag(label)
000006 F101      2    9    8         jr      label
                           .
                           .
```

LINE

This column contains the line number. This is a decimal number indicating each input line, starting from 1 and incrementing with each source line. If listing of the line is suppressed (i.e. by $LIST OFF), the number increases by one anyway.

Example:

The following source part,

```
                ;Line 12
        $LIST OFF
                ;Line 14
        $LIST ON
                ;Line 16
```

results in the following list file part:

```
ADDR   CODE      CYCLES LINE SOURCELINE
                           .
                           .
                           12           ;Line 12
                           16           ;Line 16
```

SOURCELINE

This column contains the source text. This is a copy of the source line from the source module. For ease of reading the list file, tabs are expanded with sufficient numbers of blank spaces.

If the source column in the listing is too narrow to show the whole source line, the source line is continued in the next listing line.

Errors and warnings are included in the list file following the line in which they occurred.

Example:

```
ADDR   CODE      CYCLES LINE SOURCELINE
                           .
                           .
021271 FF8F               29         dw      @coff(@caddr(300,8fffh))
as88 W172: /tmp/t.src line 29 : page number must be between 0 and FF
```

2.1.5 Debug Information

If the debug information generated by the C compiler is present in the source file, the **as88** assembler passes this information to the object file. This allows C source symbolic debugging. The **as88** assembler does not generate new debug information.

2.1.6 Instruction Set

The **as88** assembler accepts all the assembly language instruction mnemonics defined for the S1C88.

For a complete list of all instructions with mnemonics, operands, opcode format and states refer to the "S1C88 Core CPU Manual".

The following shows the precautions:

Dealing with the RETS Instruction

You have to take special care when a program contains a **rets** instruction. **rets** returns to the return address+2. Because the instruction affects the address returned to, the following situation does not work:

```
        carl  _label
        carl  _function ; 3-byte instruction
        ...
_label:
        ...
        rets            ; --> in effect, this will jump into the
                        ; middle of the 3-byte 'carl _function'
                        ; instruction (return address + 2)
```

The assembler is not capable of detecting this type of conflicts.

2.2 Software Concept

2.2.1 Introduction

Complex software projects often are divided into smaller program units. These subprograms may be written by a team of programmers in parallel, or they may be programs written for a precious development effort that are going to be reused. The **as88** assembler provides directives to subdivide a program into smaller parts, modules. Symbols can be defined local to a module, so that symbol names can be used without regard to the symbols in other modules. Code and data can be organized in separate sections. These sections can be named in such a way that different modules can implement different parts of these sections. These sections can be located in memory by the locator so that concerns about memory placement are postponed until after the assembly process. By using separate modules, a module can be changed without re-assembling the other modules. This speeds up the turnaround time during the development process.

2.2.2 Modules

Modules are the separate implementation parts of a project. Each module is defined in a separate file. A module is assembled separately from other modules. By using the INCLUDE directive common definitions and macros can be included in each module. Using the **mk88** utility the module file and include file dependencies can be specified so only the correct modules are re-assembled after changes to one of the files the modules depend upon.

2.2.2.1 Modules and Symbols

A module can use symbols defined in other modules and in the module itself. Symbols defined in a module can be local (other modules cannot access it) or global (other modules have access to it). Symbols outside of a module can be defined with the EXTERN directive. Local symbols are symbols defined by the LOCAL directive or symbols defined with an SET or EQU directive. Global symbols are either labels, or symbols explicitly defined global with the GLOBAL directive.

2.2.3 Sections

Sections are relocatable blocks of code and data. Sections are defined with the DEFSECT directive and have a name. A section may have attributes to instruct the locator to place it on a predefined starting address, in short or non-short memory or that it may be overlaid with another section. See the DEFSECT directive discussion for a complete description of all possible attributes. Sections are defined once and are activated with the SECT directive. The linker will check between different modules and emits an error message if the section attributes do not match. The linker will also concatenate all matching section definitions into one section. So, all ".text" sections generated by the compiler will be linked into one big ".text" chunk which will be located in one piece. By using this naming scheme it is possible to collect all pieces of code or data belonging together into one bigger section during the linking phase. A SECT directive referring to an earlier defined section is called a continuation. Only the name can be specified.

2.2.3.1 Section Names

The assembler generates object files in relocatable IEEE-695 object format. The assembler groups units of code and data in the object file using sections. All relocatable information is related to the start address of a section. The locator assigns absolute addresses to sections. A section is the smallest unit of code or data that can be moved to a specific address in memory after assembling a source file. The compiler requires that the assembler supports several different sections with appropriate attributes to assign specific characteristics to those sections. (section with read only data, sections with code etc.)

```
DEFSECT sect_name, sect_type [, attrib ]... [ AT address ]
```

A section must be declared before it can be used. The DEFSECT directive declares a section with its attributes. A section name can be any identifier. The '@' character is not allowed in regular section names. The assembler and linker use this character to create overlayable sections. This is explained below.

The section type can be:

sect_type :	CODE | DATA	This defines in what memory (CODE or DATA) the section is located.

The section attributes can be:

attrib :	SHORT	within first 32K of code memory or within first 64K of data memory
	FIT 100H	section must fit within one 256 byte page
	FIT 8000H	section must fit within one 32K byte page
	FIT 10000H	section must fit within one 64K byte page
	CLEAR	clear section during program startup
	NOCLEAR	section is not cleared during startup
	INIT	initialization data copied from ROM to RAM at startup
	OVERLAY	section must have an overlay name
	ROMDATA	section contains data instead of executable code
	JOIN	group sections together

Unless disabled, the startup code in the tool chain has to clear data sections with the **CLEAR** attribute. These sections contain data space allocations for which no initializers have been specified. CLEAR sections are zeroed (cleared) at program startup. Sections can be excluded from this initialization with the **NOCLEAR** attribute. This is also the default situation for all sections.

Sections with the **SHORT** attribute must be allocated in the first 32K byte of code memory of the S1C88 (i.e. for CODE sections) or within the first 64K of data memory (for DATA sections). The locator produces a warning if a section with the SHORT attribute cannot be allocated in this area.

You can group sections together with the **JOIN** attribute. For example, when more sections have to be located within the same data page, you can use this attribute.

A section becomes overlayable by specifying the **OVERLAY** attribute. Only DATA sections are overlayable. The assembler reports an error if it finds the attribute combined with sections of other types. Because it is useless to initialize overlaid sections at program startup time (code using overlaid data cannot assume that the data is in the defined state upon first use), the NOCLEAR attribute is defined implicitly when OVERLAY is specified. Overlayable section names are composed as follows:

```
DEFSECT "OVLN@nfunc", DATA, OVERLAY, SHORT
          ↑    ↑
     pool name  function name
```

The linker overlays sections with the same pool name. To decide whether DATA sections can be overlaid, the linker builds a call graph. Data in sections belonging to functions that call each other cannot be overlaid. The compiler generates pseudo instructions (CALLS) with information for the linker to build this call graph. The CALLS pseudo has the following (simplified) syntax:

```
CALLS 'caller_name', 'callee_name' [, 'callee_name' ]...
```

If the function `main()` has overlayable data allocations in short memory and calls `nfunc()`, the following sections and call information will be generated:

```
DEFSECT "OVLN@nfunc", DATA, OVERLAY, SHORT
DEFSECT "OVLN@main", DATA, OVERLAY, SHORT

CALLS 'main', 'nfunc'
```

Sections become absolute when an address has been specified in the declaration using the **AT** keyword. The assembler generates information in the object file which instructs the locator to put the section contents at the specified address. It is not allowed to make an overlayable section absolute. The assembler reports an error if the AT keyword is used in combination with the OVERLAY section attribute.

After a section has been declared, it can be activated and re-activated with the SECT directive:

```
        DEFSECT ".STRING", CODE, ROMDATA
        SECT    ".STRING"
_l001:  ASCII   "hello world"
```

All instructions and pseudos which generate data or code must be within an active section. The assembler emits a warning if code or data starts without a section definition and activation.

2.2.3.2 Absolute Sections

Absolute sections (i.e. DEFSECT directives with a start address) may only be continued in the defining module (continuation). When such a section is defined in the same manner in another module, the locator will try to place the two sections at the same address. This results in a locator error. When an absolute section is defined in more than one module, the section must be defined relocatable and its starting address must be defined in the locator description (`.dsc`) file. Overlay sections may not be defined absolute.

2.2.3.3 Grouped Sections

When you have to group sections together in one page, you can use the **JOIN** section attribute. The **JOIN** attribute should be used together with the **FIT** attribute, which defines the page size. The page size for one particular group should be the same for all sections in the group. For example, when two data sections have to be located within the same 64K page, you can write this as follows:

```
DEFSECT ".data1@group", DATA, JOIN, FIT 10000H
SECT    ".data1@group"
```

and for the second section:

```
DEFSECT ".data2@group", DATA, JOIN, FIT 10000H
SECT    ".data2@group"
```

Note that sections are grouped by the extension used in the section name. So, the definition is:

```
DEFSECT "sect@group", DATA, JOIN, FIT 10000H
          ↑     ↑
    section name   joined group name
```

2.2.3.4 Section Examples

Some examples of the DEFSECT and SECT directives are as follows:

```
DEFSECT ".CONST", CODE AT 1000H
SECT    ".CONST"
```

Defines and activates a section named `.CONST` starting on address 1000H. Other parts of the same section, and in the same module, must be defined with:

```
SECT    ".CONST"
```

```
DEFSECT ".text", CODE
SECT    ".text"
```

Defines and activates a relocatable section in CODE memory. Other parts of this section, with the same name, may be defined in the same module or any other module. Other modules should use the same DEFSECT statement. When necessary, it is possible to give the section an absolute starting address with the locator description file.

```
DEFSECT ".fardata", DATA, CLEAR
SECT    ".fardata"
```

Defines a relocatable named section in DATA memory. The CLEAR attribute instructs the locator to clear the memory located to this section. When this section is used in another module it must be defined identically.

Continuations of this section in the same module are as follows:

```
SECT    ".fardata"
```

```
DEFSECT ".ovlf@f", DATA, OVERLAY
SECT    ".ovlf@f"
```

Defines a relocatable section in DATA memory. The section may be overlaid with other overlayable DATA sections. The function associated with this overlayable part is "f". This is the name that should be used with the CALLS directive to designate which function call each other so the linker can build a correct call graph. See also Section 2.2.3.1, "Section Names".

2.3 Assembly Language

2.3.1 Input Specification

An assembly program consists of zero or more statements, one statement per line. A statement may optionally be followed by a comment, which is introduced by a semicolon character (;) and terminated by the end of the input line. Any source statement can be extended to one or more lines by including the line continuation character (\) as the last character on the line to be continued. The length of a source statement (first line and any continuation lines) is only limited by the amount of available memory. Upper and lower case letters are considered equivalent for assembler mnemonics and directives, but are considered distinct for labels, symbols, directive arguments, and literal strings.

A *statement* can be defined as:

[*label*:] [*instruction* | *directive* | *macro_call*] [;*comment*]

where,

label is an *identifier*. A label does not have to start on the first position of a line, but a label must always be followed by a colon.
identifier can be made up of letters, digits and/or underscore characters (_). The first character may not be a digit. The size of an identifier is only limited by the amount of available memory.

Example:

```
LAB1:               ; This is a label
```

instruction is any valid S1C88 assembly language instruction consisting of a mnemonic and operands. Operands are described in Section 2.4, "Operands and Expressions". See the "S1C88 Core CPU Manual" for details of the instructions.

Examples:

```
RET                 ; No operand
PUSH    A           ; One operand
ADD     BA,HL       ; Two operands
```

directive any one of the assembler directives; described separately in Section 2.6, "Assembler Directives".

macro_call a call to a previously defined macro. See Section 2.5, "Macro Operations".

A statement may be empty.

2.3.2 Assembler Significant Characters

There are several one character sequences that are significant to the assembler. Some have multiple meanings depending on the context in which they are used. Special characters associated with expression evaluation are described in Section 2.4, "Operands and Expressions". Other assembler-significant characters are:

- **;** - Comment delimiter
- **** - Line continuation character or
  Macro dummy argument concatenation operator
- **?** - Macro value substitution operator
- **%** - Macro hex value substitution operator
- **^** - Macro local label operator
- **"** - Macro string delimiter or
  Quoted string **DEFINE** expansion character
- **@** - Function delimiter
- ***** - Location counter substitution
- **[]** - Location addressing mode operator
- **#** - Immediate addressing mode operator

Individual descriptions of each of the assembler special characters follow. They include usage guidelines, functional descriptions, and examples.

;

Comment Delimiter Character

Any number or characters preceded by a semicolon (;), but not part of a literal string, is considered a comment. Comments are not significant to the assembler, but they can be used to document the source program. Comments will be reproduced in the assembler output listing. Comments are preserved in macro definitions.

Comments can occupy an entire line, or can be placed after the last assembler-significant field in a source statement. The comment is literally reproduced in the listing file.

Examples:

```
; This comment begins in column 1 of the source file
Loop:   CALL    [COMPUTE]       ; This is a trailing comment
        ; These two comments are preceded
        ; by a tab in the source file
```

\

Line Continuation Character or Macro Dummy Argument Concatenation Operator

Line Continuation

The backslash character (\), if used as the last character on a line, indicates to the assembler that the source statement is continued on the following line. The continuation line will be concatenated to the previous line of the source statement, and the result will be processed by the assembler as if it were a single line source statement. The maximum source statement length (the first line and any continuation lines) is 512 characters.

Example:

```
; THIS COMMENT \
EXTENDS OVER \
THREE LINES
```

Macro Argument Concatenation

The backslash (\) is also used to cause the concatenation of a macro dummy argument with other adjacent alphanumeric characters. For the macro processor to recognize dummy arguments, they must normally be separated from other alphanumeric characters by a non-symbol character. However, sometimes it is desirable to concatenate the argument characters with other characters. If an argument is to be concatenated in front of or behind some other symbol characters, then it must be followed by or preceded by the backslash, respectively.

See also Section 2.5.5.1, "Dummy Argument Concatenation Operator - \".

Example:

Suppose the source input file contained the following macro definition:

```
SWAP_MEM MACRO  REG1,REG2       ;swap memory contents
         LD     A,[I\REG1]      ;using A as temp
         LD     B,[I\REG2]      ;using B as temp
         LD     [I\REG1],B
         LD     [I\REG2],A
         ENDM
```

The concatenation operator (\) indicates to the macro processor that the substitution characters for the dummy arguments are to be concatenated in both cases with the character I. If this macro were called with the following statement,

```
         SWAP_MEM        X,Y
```

the resulting expansion would be:

```
         LD      A,[IX]
         LD      B,[IY]
         LD      [IX],B
         LD      [IY],A
```

?

Return Value of Symbol Character

The **?***symbol* sequence, when used in macro definitions, will be replaced by an ASCII string representing the value of *symbol*. This operator may be used in association with the backslash (\) operator. The value of *symbol* must be an integer.

See also Section 2.5.5.2, "Return Value Operator - ?".

Example:

Consider the following macro definition:

```
SWAP_MEM MACRO  REG1,REG2       ;swap memory contents
         LD     A,[_lab\?REG1]  ;using A as temp
         LD     B,[_lab\?REG2]  ;using B as temp
         LD     [_lab\?REG1],B
         LD     [_lab\?REG2],A
         ENDM
```

If the source file contained the following SET statements and macro call,

```
AREG     SET      1
BREG     SET      2
         SWAP_MEM AREG,BREG
```

the resulting expansion as it would appear on the source listing would be:

```
         LD      A,[_lab1]
         LD      B,[_lab2]
         LD      [_lab1],B
         LD      [_lab2],A
```

%

Return Hex Value of Symbol Character

The **%***symbol* sequence, when used in macro definitions, will be replaced by an ASCII string representing the hexadecimal value of *symbol*. This operator may be used in associations with the backslash (\) operator. The value of *symbol* must be an integer.

See also Section 2.5.5.3, "Return Hex Value Operator - %".

Example:

Consider the following macro definition:

```
GEN_LAB   MACRO LAB,VAL,STMT
LAB\%VAL: STMT
          ENDM
```

If this macro were called as follows,

```
NUM     SET     10
        GEN_LAB HEX,NUM,'NOP'
```

The resulting expansion as it would appear in the listing file would be:

```
HEXA:   NOP
```

^

Macro Local Label Character

The circumflex (^), when used as a unary operator in a macro expansion, will cause name mangling of any associated local label. Normally, the macro preprocessor will leave any local label inside a macro expansion to a normal label in the current module. By using the Local Label character (^), the label is made a unique label. This is done by removing the leading underscore and appending a unique string "__M_L*xxxxxx*" where "*xxxxxx*" is a unique sequence number. The ^-operator has no effect outside of a macro expansion. The ^-operator is useful for passing label names as macro arguments to be used as local label names in the macro. Note that the circumflex is also used as the binary exclusive or operator.

See also Section 2.5.5.5, "Macro Local Label Operator - ^".

Example:

Consider the following macro definition:

```
LOAD    MACRO   ADDR
        ADDR:
                LD      A,[ADDR]
        ^ADDR:
                LD      A,[^ADDR]
        ENDM
```

If this macro were called as follows,

```
        LOAD    _LOCAL
```

the resulting expansion as it would appear in the listing file would be:

```
_LOCAL:
        LD      A,[_LOCAL]
_LOCAL__M_L000001:
        LD      A,[_LOCAL__M_L000001]
```

"

Macro String Delimiter or Quoted String DEFINE Expansion Character

Macro String

The double quote ("), when used in macro definitions, is transformed by the macro processor into the string delimiter, the single quote ('). The macro processor examines the characters between the double quotes for any macro arguments. This mechanism allows the use of macro arguments as literal strings.

See also Section 2.5.5.4, "Dummy Argument String Operator - "".

Example:

Using the following macro definition,

```
CSTR    MACRO   STRING
        ASCII   "STRING"
        ENDM
```

and a macro call,

```
        CSTR    ABCD
```

the resulting macro expansion would be:

```
        ASCII   'ABCD'
```

Quoted String DEFINE Expansion

A sequence of characters which matches a symbol created with a **DEFINE** directive will not be expanded if the character sequence is contained within a quoted string. Assembler strings generally are enclosed in single quotes ('). If the string is enclosed in double quotes (") then **DEFINE** symbols will be expanded within the string. In all other respects usage of double quotes is equivalent to that of single quotes.

Example:

Consider the source fragment below:

```
        DEFINE  LONG    'short'
STR_MAC MACRO   STRING
        MSG     'This is a LONG STRING'
        MSG     "This is a LONG STRING"
        ENDM
```

If this macro were invoked as follows,

```
        STR_MAC sentence
```

then the resulting expansion would be:

```
        MSG     'This is a LONG STRING'
        MSG     'This is a short sentence'
```

@

Function Delimiter

All assembler built-in functions start with the **@** symbol. See Section 2.4.4, "Functions", for a full discussion of these functions.

Example:

```
SVAL    EQU     @ABS(VAL)       ; Obtain absolute value
```

*****

Location Counter Substitution

When used as an operand in an expression, the asterisk represents the current integer value of the runtime location counter.

Example:

```
          DEFSECT ".CODE", CODE  AT 100H
          SECT    ".CODE"
  XBASE   EQU     *+20H   ; XBASE = 120H
```

[]

Location Addressing Mode Operator

Square brackets are used to indicate to the assembler to use a location addressing mode.

Example:

```
          LD      A,[_Value]
```

#

Immediate Addressing Mode

The pound sign (**#**) is used to indicate to the assembler to use the immediate addressing mode.

Example:

```
  CNST    EQU     5H
          LD      A,#CNST ;Load A with the value 5H
```

2.3.3 Registers

The following S1C88 register names, either upper or lower case, cannot be used as symbol names in an assembly language source file:

```
A       BR
B       IX
BA      IY
H
L
HL

NB      SC
EP      PC
XP      SP
YP
```

2.3.4 Other Special Names

The following names, used in the S1C88 instruction set, either upper or lower case, cannot be used as symbol names in an assembly language source file:

```
C       P
T       M
LT      F0
LE      F1
GT      F2
GE      F3
V       NF0
NV      NF1
NC      NF2
NT      NF3
```

2.4 Operands and Expressions

2.4.1 Operands

An operand is the part of the instruction that follows the instruction opcode. There can be one or two or even no operands in an instruction. An operand of an assembly instruction has one of the following types:

Operands	Description
expr	any valid expression as described in the section *Expressions*.
reg	any valid register as described in the section *Registers*.
symbol	a symbolic name as created by an equate. A symbol can be an expression.
address	a combination of expr, reg and symbol.

If an expression can be completely evaluated at assembly time, it is called an absolute expression; if it is not, it is called a relocatable expression. See Section 2.4.2, "Expressions", for more details.

2.4.1.1 Operands and Addressing Modes

The S1C88 assembly language has several addressing modes. These are listed below with a short description. For details see the "S1C88 Core CPU Manual".

Register Direct

The instruction specifies the register which contains the operand.

Syntax: *mnemonic* *register*

Register Indirect

The instruction specifies the register containing the operand address. Several forms are available.

Syntax: *mnemonic* [RR]
mnemonic [RR + *off*]
mnemonic [RR + L]

Immediate

An immediate operand is a one byte number or one word number, which is encoded as part of the instruction. Immediate operands are indicated by the # sign before the expression defining the value of the operand.

Syntax: *mnemonic* *#number*

Absolute

The instruction contains the operand address. The address can be 8 or 16 bits.

Syntax: *mnemonic* [*direct_address*]

PC Relative

The instruction contains the 8 or 16-bit offset relative to the current PC value.

Syntax: *mnemonic* *offset*

Implied

The instruction implicitly defines the used registers.

Syntax: *mnemonic*

2.4.2 Expressions

An operand of an assembler instruction or directive is either an assembler symbol, a register name or an expression. An expression is a sequence of symbols that denotes an address in a particular memory space or a number.
Expressions that can be evaluated at assembly time are called **absolute expressions**. Expressions where the result is unknown until all sections have been combined and located are called **relocatable expressions**. When any operand of an expression is relocatable the entire expression is relocatable. Relocatable expressions are emitted in the object file and evaluated by the linker or the locator. Relocatable expressions may only contain integral functions. An error is emitted when during object creation non-IEEE relocatable expressions are found.
An expression has a type which depends on the type of the identifiers in the expression. See Section 2.4.2.4, "Expression Type", for details.
The assembler evaluates expressions with 64-bit precision in two's complement.
The syntax of an *expression* can be any of the following:

- *number*
- *expression_string*
- *symbol*
- *expression binary_operator expression*
- *unary_operator expression*
- *(expression)*
- *function*

All types of expressions are explained below and in the following sections.

( ) You can use parentheses to control the evaluation order of the operators. What is between parentheses is evaluated first.

Examples:

```
(3+4)*5         ; Result is 35. 3 + 4 is evaluated first.
3+(4*5)         ; Result is 23. 4 * 5 is evaluated first.
                ; parentheses are superfluous here
```

2.4.2.1 Number

Numeric constants can be used in expressions. If there is no postfix, the assembler assumes the number is in the default RADIX. The default RADIX on its turn is decimal.

number can be one of the following:
- *bin_num*B
- *dec_num* (or *dec_num*D)
- *oct_num*O (or *oct_num*Q)
- *hex_num*H

Lowercase equivalences are allowed: b, d, o, q, h.

bin_num is a binary number formed of '0'–'1' ending with a 'B' or 'b'.
Examples: `1001B; 1011B; 01100100b;`

dec_num is a decimal number formed of '0'–'9', optionally followed by the letter 'D' or 'd'.
Examples: `12; 5978D;`

oct_num is an octal number formed of '0'–'7' ending with an 'O', 'o', 'Q' or 'q'.
Examples: `11O; 447o; 30146q`

hex_num is a hexadecimal number formed of the characters '0'–'9' and 'a'–'f' or 'A'–'F' ending with a 'H' or 'h'. The first character must be a decimal digit, so it may be necessary to prefix a hexadecimal number with the '0' character.
Examples: `45H; 0FFD4h; 9abcH`

A number may be written without a following radix indicator if the input radix is changed using the RADIX directive. For example, a hexadecimal number may be written without the suffix **H** if the input radix is set to 16 (assuming an initial radix of 10). The default radix is 10.

2.4.2.2 Expression String

An *expression_string* is a *string* with an arbitrary length evaluating to a number. The value of the string is calculated by taking the first 4 characters padded with 0 to the left.

string is a string of ASCII characters, enclosed in single (') or double (") quotes. The starting and closing quote must be the same. To include the enclosing quote in the string, double it. E.g. the string containing both quotes can be denoted as: " ' "" " or ' ' ' " '.

See Section 2.5, "Macro Operations", for the differences between single and double quoted strings.

Examples:

```
'A'+1   ; a 1-character ASCII string,
        ; result 42H
"9C"+1  ; a 2-character ASCII string,
        ; result 3944H
```

2.4.2.3 Symbol

A *symbol* is an *identifier*. A *symbol* represents the value of an *identifier* which is already defined, or will be defined in the current source module by means of a label declaration or an equate directive.

Examples:

```
CON1 EQU 3H     ; The variable CON1 represents the value of 3

LD A,[CON1+20H] ; Load A with contents of address 23H
```

When you invoke the assembler, the following predefined symbols exist:

_AS88 contains a string with the name of the assembler ("as88")

_MODEL contains an integer with the ASCII value of the selected MODEL (in lower case)

2.4.2.4 Expression Type

The type of an expression is either a number (integral) or an address. The result type of an expression depends on the operator and its operands. The tables below summarize all available operators.

Please note:

1. a label is of type 'address'; an equate symbol has the type of the equate expression;
2. the type of an untyped symbol can be an address or a number, depending on the context; the result of the operation can be determined using the tables;
3. the binary logical and relational operators (||, &&, ==, !=, <, <=, >, >=) accept any combination of operands, the result is always the integral number 0 or 1;
4. the binary shift and bitwise operators <<, >>, |, & and ^ only accept integral operands.

The following table shows the result type of expressions with unary operators (a '*' indicates an illegal combination).

Table 2.4.2.4.1 Expression type, unary operators

Operator	integer	addr
~	integer	*
!	integer	*
-	integer	*
+	integer	integer

The following table shows the result type of expressions with binary numerical operators.

Table 2.4.2.4.2 Expression type, binary numerical operators

Operator	integer, integer	addr, integer	integer, addr	addr, addr
-	integer	addr	*	integer
+	integer	addr	addr	*
*	integer	*	*	*
/	integer	*	*	*
%	integer	*	*	*

Note: A string operand will be converted to an integral number.

The following table shows the result type of functions. A '–' in the column Operands means that the function has no operands.

Table 2.4.2.4.3 Expression type, functions

Function	Operands	Result
@ABS()	integer	integer
@ARG()	symbol integer	integer integer
@AS88()	–	string
@CADDR()	integer,addr	addr
@CAT()	string,string	string
@CNT()	–	integer
@COFF()	addr	addr
@CPAG()	addr	integer
@DADDR()	integer,addr	addr
@DEF()	symbol	integer
@DOFF()	addr	addr
@DPAG()	addr	integer
@HIGH()	addr	integer
@LEN()	string	integer
@LOW()	addr	integer
@LST()	–	integer
@MAC()	symbol	integer
@MAX()	integer,integer,...	integer
@MIN()	integer,integer,...	integer
@MODEL()	–	integer
@MXP()	–	integer
@POS()	string,string string,string,integer	integer integer
@SCP()	string,string	integer
@SGN()	integer	integer
@SUB()	string,integer,integer	string

2.4.3 Operators

There are two types of operators:

- unary operators
- binary operators

Operators can be arithmetic operators, shift operators, relational operators, bitwise operators, or logical operators. All operators are described in the following sections.

If the grouping of the operators is not specified with parentheses, the operator precedence is used to determine evaluation order. Every operator has a precedence level associated with it. The following table lists the operators and their order of precedence (in descending order).

Table 2.4.3.1 Operators precedence list

Operators	Type
+, -, ~, !	unary
*, /, %	binary
+, -	binary
<<, >>	binary
<, <=, >, >=	unary
==, !=	binary
&	binary
^	binary
|	binary
&&	binary
||	binary

Except for the unary operators, the assembler evaluates expressions with operators of the same precedence level left-to-right. The unary operators are evaluated right-to-left. So, `-4+3*2` evaluates to `(-4)+(3*2)`.

2.4.3.1 Addition and Subtraction

Synopsis:

Addition: *operand* **+** *operand*

Subtraction: *operand* **-** *operand*

The + operator adds its two operands and the - operator subtracts them. The operands can be any expression evaluating to an absolute number or a relocatable operand, with the restrictions of Table 2.4.2.4.2.

Examples:

```
0A342H + 23H    ; addition of absolute numbers
0FF1AH - AVAR   ; subtraction with the value of symbol AVAR
```

2.4.3.2 Sign Operators

Synopsis:

Plus: **+***operand*

Minus: **-***operand*

The + operator does not modify its operand. The - operator subtracts its operand from zero. See also the restrictions in Table 2.4.2.4.1.

Example:

```
5+-3    ; result is 2
```

2.4.3.3 Multiplication and Division

Synopsis:

Multiplication: *operand* ***** *operand*

Division: *operand* **/** *operand*

Modulo: *operand* **%** *operand*

The * operator multiplies its two operands, the / operator performs an integer division, discarding any remainder. The % operator also performs an integer division, but discards the quotient and returns the remainder. The operands can be any expression evaluating to an absolute number or a relocatable operand, with the restrictions of Table 2.4.2.4.2. Note that the right operands of the / and % operator may not be zero.

Examples:

```
AVAR*2          ; multiplication
0FF3CH/COUNT    ; division
23%4            ; modulo, result is 3
```

2.4.3.4 Shift Operators

Synopsis:

Shift left: *operand* << *count*

Shift right: *operand* >> *count*

These operators shift their left operand (*operand*) either left (<<) or right (>>) by the number of bits (absolute number) specified with the right operand (*count*). The operands can be any expression evaluating to an (integer) number.

Example:

```
AVAR>>4         ; shift right variable AVAR, 4 times
```

2.4.3.5 Relational Operators

Synopsis:

Equal: *operand* == *operand*

Not equal: *operand* != *operand*

Less than: *operand* < *operand*

Less than or equal: *operand* <= *operand*

Greater than: *operand* > *operand*

Greater than or equal: *operand* >= *operand*

These operators compare their operands and return an absolute number (an integer) of 1 for 'true' and 0 for 'false'. The operands can be any expression evaluating to an absolute number or a relocatable operand.

Examples:

```
3>=4            ; result is 0 (false)
4==COUNT        ; 1 (true), if COUNT is 4.
                ; 0 otherwise.
9<0AH           ; result is 1 (true)
```

2.4.3.6 Bitwise Operators

Synopsis:

Bitwise AND:	*operand*	**&**	*operand*
Bitwise OR:	*operand*	|	*operand*
Bitwise XOR:	*operand*	^	*operand*
One's complement:		**~**	*operand*

The AND, OR and XOR operators take the bitwise AND, OR respectively XOR of the left and right operand. The one's complement (bitwise NOT) operator performs a bitwise complement on its operand. The operands can be any expression evaluating to an (integer) number.

Examples:

```
0BH&3   ; result is 3
                1011B
                0011B  &
                0011B

~0AH    ; result is 0FFF5H
                ~ 00000000 00001010B
                = 11111111 11110101B
```

2.4.3.7 Logical Operators

Synopsis:

Logical AND:	*operand*	**&&**	*operand*
Logical OR:	*operand*	||	*operand*
Logical NOT:		**!**	*operand*

The logical AND operator returns an integer 1 if both operands are non-zero; otherwise it returns an integer 0. The logical OR operator returns an integer 1 if either of its operands is non-zero; otherwise it returns an integer 0. The **!** operator performs a logical not on its operand. **!** returns an integer 1 ('true) if the operand is 0; otherwise, **!** returns 0 ('false'). The operands can be can be any expression evaluating to an integer.

Examples:

```
0BH&&3  ; result is 1 (true)

!0AH    ; result is 0 (false)
!(4<3)  ; result is 1 (true)
        ; 4 < 3 result is 0 (false)
```

2.4.4 Functions

The assembler has several built-in functions to support data conversion, string comparison, and math computations. Functions can be used as terms in any arbitrary expression. Functions have the following syntax:

```
@function_name(argument[,argument]...)
```

Functions start with the '@' sign and have zero or more arguments, and are always followed by opening and closing parentheses. There must be no intervening spaces between the function name and the opening parenthesis and between the (comma-separated) arguments.

Assembler functions can be grouped into five types:

1. Mathematical functions
2. String functions
3. Macro functions
4. Assembler mode functions
5. Address handling functions

2.4.4.1 Mathematical Functions

The mathematical functions comprise min/max functions, among others:

ABS - Absolute value

MAX - Maximum value

MIN - Minimum value

SGN - Return sign

2.4.4.2 String Functions

String functions compare strings, return the length of a string, and return the position of a substring within a string:

CAT - Catenate strings

LEN - Length of string

POS - Position of substring in string

SCP - Compare strings

SUB - Substring from a string

2.4.4.3 Macro Functions

Macro functions return information about macros:

ARG - Macro argument function

CNT - Macro argument count

MAC - Macro definition function

MXP - Macro expansion function

2.4.4.4 Assembler Mode Functions

Miscellaneous functions having to do with assembler operation:

AS88 - Assembler executable name

DEF - Symbol definition function

LST - LIST control flag value

MODEL - Selected model of the assembler

2.4.4.5 Address Handling Functions

Functions handling specific address arithmetic:

CADDR - Code address
COFF - Code page offset
CPAG - Code page number
DADDR - Data address
DOFF - Data page offset
DPAG - Data page number
HIGH - 256 byte page number
LOW - 256 byte page offset

2.4.4.6 Detailed Description

Individual descriptions of each of the assembler functions follow. They include usage guidelines, functional descriptions, and examples.

@ABS(*expression*)

Returns the absolute value of *expression* as an integer value.

Example:

```
LD      A,#@ABS(VAL)            ;load absolute value
```

@ARG(*symbol* | *expression*)

Returns integer 1 if the macro argument represented by *symbol* or *expression* is present, 0 otherwise. If the argument is a symbol it must be single-quoted and refer to a dummy argument name. If the argument is an expression it refers to the ordinal position of the argument in the macro dummy argument list. A warning will be issued if this function is used when no macro expansion is active.

Example:

```
IF      @ARG('TWIDDLE')         ;twiddle factor provided?
```

@AS88()

Returns the name of the assembler executable. This is **as88** for the S1C88 family.

Example:

```
ANAME:  DB      @AS88()         ;ANAME = 'as88'
```

@CADDR(*code-page*,*code-offset*)

Returns the code address specified by the code page (32K bank) and page offset (32K offset). The resulting value will be a relocatable expression when *code-offset* is relocatable. When *code-offset* is absolute, the result will be a constant value.

Example:

```
CAZERO  SET     @CADDR(3,8004h) ;CAZERO = 18004h
CAONE   SET     @CADDR(3,5000h) ;CAONE  =  5000h
```

@CAT(*str1*,*str2*)

Concatenates the two strings into one string. The two strings must be enclosed with single or double quotes.

Example:

```
DEFINE  ID      "@CAT('S1C','88')"      ;ID = 'S1C88'
```

@CNT()

Returns the count of the current macro expansion arguments as an integer. A warning will be issued if this function is used when no macro expansion is active.

Example:

```
ARGCNT  SET     @CNT()          ;squirrel away arg count
```

@COFF(*address*)

Returns the code page offset (32K offset) of the given address. The resulting value will be a relocatable expression when *address* is relocatable. When *address* is absolute, the result will be a constant value. Bit 16 (MSB) of the result is 0 when the address is in the first code page (first 32K). In all other cases bit 16 (MSB) of the result is 1.

Example:

```
PAGEZERO  SET   @COFF(07FFFH)   ;PAGEZERO = 07FFFH
PAGEONE   SET   @COFF(0CFFFH)   ;PAGEONE  = 0CFFFH
PAGETWO   SET   @COFF(014FFFH)  ;PAGETWO  = 0CFFFH
```

@CPAG(*address*)

Returns the code page (32K bank) of the given address. The resulting value will be a relocatable expression when *address* is relocatable. When *address* is absolute, the result will be a constant value.

Example:

```
ZEROPAGE  SET   @CPAG(07FFFH)   ;ZEROPAGE = 0H
ONEPAGE   SET   @CPAG(0CFFFH)   ;ONEPAGE  = 1H
TWOPAGE   SET   @CPAG(014FFFH)  ;TWOPAGE  = 2H
```

@DADDR(*data-page*,*data-offset*)

Returns the data address specified by the data page (64K bank) and page offset (64K offset). The resulting value will be a relocatable expression when *data-offset* is relocatable. When *data-offset* is absolute, the result will be a constant value.

Example:

```
DATHREE SET     @DADDR(3,1234h) ;DATHREE = 31234h
```

@DEF(*symbol*)

Returns an integer 1 if *symbol* has been defined, 0 otherwise. *symbol* may be any label not associated with a **MACRO** directive. If *symbol* is quoted it is looked up as a **DEFINE** symbol; if it is not quoted it is looked up as an ordinary label.

Example:

```
IF      @DEF('ANGLE')           ;assemble if ANGLE defined
```

@DOFF(*address*)

Returns the data page offset (64K offset) of the given address. The resulting value will be a relocatable expression when *address* is relocatable. When *address* is absolute, the result will be a constant value.

Example:

```
PAGEZERO  SET   @DOFF(07FFFH)   ;PAGEZERO = 07FFFH
PAGEONE   SET   @DOFF(0CFFFH)   ;PAGEONE  = 0CFFFH
PAGETWO   SET   @DOFF(014FFFH)  ;PAGETWO  = 04FFFH
```

@DPAG(*address*)

Returns the data page (64K bank) of the given address. The resulting value will be a relocatable expression when *address* is relocatable. When *address* is absolute, the result will be a constant value.

Example:

```
ZEROPAGE  SET   @DPAG(07FFFH)   ;ZEROPAGE = 0H
ONEPAGE   SET   @DPAG(0CFFFH)   ;ONEPAGE  = 0H
TWOPAGE   SET   @DPAG(014FFFH)  ;TWOPAGE  = 1H
```

@HIGH(*address*)

Returns the 256 byte page number of the given address. The resulting value will be a relocatable expression when *address* is relocatable. When *address* is absolute, the result will be a constant value.

Example:

```
HPAGE   SET     @HIGH(07FFFH)   ;HPAGE = 07FH
```

@LEN(*string*)

Returns the length of *string* as an integer.

Example:

```
SLEN    SET     @LEN('string')  ;SLEN = 6
```

@LOW(*address*)

Returns the 256 byte page offset of the given address. The resulting value will be a relocatable expression when *address* is relocatable. When *address* is absolute, the result will be a constant value.

Example:

```
LPAGE   SET     @LOW(07FFFH)    ;LPAGE = 0FFH
```

@LST()

Returns the value of the **LIST** control flag as an integer. Whenever a **LIST ON** control is encountered in the assembler source, the flag is incremented; when a **LIST OFF** control is encountered, the flag is decremented.

Example:

```
DUP     @ABS(@LST())            ;list unconditionally
```

@MAC(*symbol*)

Returns an integer 1 if *symbol* has been defined as a macro name, 0 otherwise.

Example:

```
IF      @MAC(DOMUL)             ;expand macro
```

@MAX(*expr1*[,*exprN*]...)

Returns the greatest of *expr1*, ..., *exprN* as an integer.

Example:

```
MAX:    DB      @MAX(1,5,-3)    ;MAX = 5
```

@MIN(*expr1*[,*exprN*]...)

Returns the least of *expr1, ..., exprN* as an integer.

Example:

```
MIN:    DB      @MIN(1,5,-3)    ;Min = -3
```

@MODEL()

Returns the selected model of the assembler (as specified on the command line or as specified with the MODEL control). The returned value is the ASCII character value of the selected MODEL (always in lower case).

Example (assumes **-Ms** option):

```
MDL     SET     @MODEL()        ;MDL = 73h (ASCII value of 's')
```

@MXP()

Returns an integer 1 if the assembler is expanding a macro, 0 otherwise.

Example:

```
IF      @MXP()                  ;macro expansion active?
```

@POS(*str1*,*str2*[,*start*])

Returns the position *str2* in *str1* as an integer, starting at position *start*. If *start* is not given the search begins at the beginning of *str1*. If the *start* argument is specified it must be a positive integer and cannot exceed the length of the source string.

Example:

```
ID      EQU     @POS('S1C88','88')      ;ID = 3
```

@SCP(*str1*,*str2*)

Returns an integer 1 if the two strings compare, 0 otherwise. The two strings must be separated by a comma.

Example:

```
IF      @SCP(STR,'MAIN')        ;does STR equal MAIN?
```

@SGN(*expression*)

Returns the sign of *expression* as an integer: -1 if the argument is negative, 0 if zero, 1 if positive. The *expression* may be relative or absolute.

Example:

```
IF      @SGN(INPUT) == 1        ;is sign positive?
```

@SUB(*str*,*expr1*,*expr2*)

Returns the substring from *str* as a string. *expr1* is the starting position within *str* and *expr2* is the length of the desired string. The assembler issues an error if either *expr1* or *expr2* exceeds the length of *str*.

Example:

```
DEFINE  ID      "@SUB('S1C88',3,2)"     ;ID = '88'
```

2.5 Macro Operations

2.5.1 Introduction

This chapter describes the macro operations and conditional assembly.
The macro preprocessor is implemented in the assembler.

2.5.2 Macro Operations

Programming applications frequently involve the coding of a repeated pattern or group of instructions. Some patterns contain variable entries which change for each repetition of the pattern. Others are subject to conditional assembly for a given occurrence of the instruction group. In either case, macros provide a shorthand notation for handling these instruction patterns. Having determined the iterated pattern, the programmer can, within the macro, designate selected fields of any statement as variable. Thereafter by invoking a macro the programmer can use the entire pattern as many times as needed, substituting different parameters for the designated variable portions of the statements.

When the pattern is defined it is given a name. This name becomes the mnemonic by which the macro is subsequently invoked (called). If the name of the macro is the same as an existing assembler directive or mnemonic opcode, the macro will replace the directive or mnemonic opcode, and a warning will be issued.

The macro call causes source statements to be generated. The generated statements may contain substitutable arguments. The statements produced by a macro call are relatively unrestricted as to type. They can be any processor instruction, almost any assembler directive, or any previously-defined macro. Source statements resulting from a macro call are subject to the same conditions and restrictions that are applied to statements written by the programmer.

To invoke a macro, the macro name must appear in the operation code field of a source statement. Any arguments are placed in the operand field. By suitably selecting the arguments in relation to their use as indicated by the macro definition, the programmer causes the assembler to produce in-line coding variations of the macro definition.

The effect of a macro call is to produce in-line code to perform a predefined function. The code is inserted in the normal flow of the program so that the generated instructions are executed with the rest of the program each time the macro is called.

An important feature in defining a macro is the use of macro calls within the macro definition. The assembler processes such **nested** macro calls at expansion time only. The nesting of one macro definition within another definition is permitted. However, the nested macro definition will not be processed until the primary macro is expanded. The macro must be defined before its appearance in a source statement operation field.

2.5.3 Macro Definition

The definition of a macro consists of three parts: the header, which assigns a name to the macro and defines the dummy arguments; the body, which consists of prototype or skeleton source statements; and the terminator. The header is the **MACRO** directive, its name, and the dummy argument list. The body contains the pattern of standard source statements. The terminator is the **ENDM** directive.

The header of a macro definition has the form:

macro_name **MACRO** [*dummy argument list*] [*comment*]

The required name is the symbol by which the macro will be called. The dummy argument list has the form:

[*dumarg*[,*dumarg*]...]

The dummy arguments are symbolic names that the macro processor will replace with arguments when the macro is expanded (called). Each dummy argument must obey the same rules as global symbol names. Dummy arguments are separated by commas.

When a macro call is executed, the dummy arguments within the macro definition (`NMUL`, `AVEC`, `BVEC`, `OFFSET`, `RESULT` in the example below) are replaced with the corresponding argument as defined by the macro call.

All local label definitions within a macro which use the local label operator are made unique for this macro call. This is done by appending a unique postfix to every local label, making the scope of the label local to the module. This mechanism allows the programmer to freely use local labels within a macro definition without regard to the number of times that the macro is expanded. Labels without the local label operator are considered to be normal labels and thus cannot occur more than once unless used with the **SET** directive (see Section 2.6, "Assembler Directives").

Example:

The macro:

```
N_R_MUL MACRO   NMUL,AVEC,BVEC,OFFSET,RESULT   ;header
        LD      B,#NMUL                        ;body
        LD      IX,#AVEC
        LD      IY,#BVEC
^again: LD      L,[IX+OFFSET]
        LD      A,[IY+OFFSET]
        MLT
        ADD     A,[RESULT]
        LD      [RESULT],A
        INC     IX
        INC     IY
        DJR     NZ,^again
        ENDM                                   ;terminator
        N_R_MUL 10H,_obj1,_obj2,10H,_RESULT
```

expands to: (note the different handling of `again` and `_RESULT`)

```
        LD      B,#10H
        LD      IX,#_obj1
        LD      IY,#_obj2
again__M_L000001:
        LD      L,[IX+10H]
        LD      A,[IY+10H]
        MLT
        ADD     A,[_RESULT]
        LD      [_RESULT],A
        INC     IX
        INC     IY
        DJR     NZ,again__M_L000001
```

2.5.4 Macro Calls

When a macro is invoked the statement causing the action is termed a **macro call**. The syntax of a macro call consists of the following fields:

[*label*:] *macro_name* [*arguments*] [*comment*]

The argument field can have the form:

[*arg*[,*arg*]...]

The macro call statement is made up of three besides the comment field: the *label*, if any, will correspond to the value of the location counter at the start of the macro expansion; the operation field which contains the macro name; and the operand field which contains substitutable arguments. Within the operand field each calling argument of a macro call corresponds one-to-one with a dummy argument of the macro definition. For example, the N_R_MUL macro defined earlier could be invoked for expansion (called) by the statement:

```
N_R_MUL   CNT+1,VEC1,VEC2,OFFS,OUT
```

where the operand field arguments, separated by commas and taken left to right, correspond to the dummy arguments `NMUL` through `RESULT`, respectively. These arguments are then substituted in their corresponding positions of the definition to produce a sequence of instructions.

Macro arguments consist of sequences of characters separated by commas. Although these can be specified as quoted strings, to simplify coding the assembler does not require single quotes around macro argument strings. However, if an argument has an embedded comma or space, that argument must be surrounded by single quotes ('). An argument can be declared null when calling a macro. However, if must be declared explicitly null. Null arguments can be specified in four ways:

- by writing the delimiting commas in succession with no intervening spaces;
- by terminating the argument list with a comma and omitting the rest of the argument list;
- by declaring the argument as a null string;
- by simply omitting some or all of the arguments.

A null argument will cause no character to be substituted in the generated statements that reference the argument. If more arguments are supplied in the macro call than appear in the macro definition, a warning will be issued by the assembler.

2.5.5 Dummy Argument Operators

The assembler macro processor provides for text substitution of arguments during macro expansion. In order to make the argument substitution facility more flexible, the assembler also recognizes certain text operators within macro definitions which allow for transformations of the argument text. These operators can be used for text concatenation, numeric conversion, and string handling.

2.5.5.1 Dummy Argument Concatenation Operator - \

Dummy arguments that are intended to be concatenated with other characters must be preceded by the concatenation operator, '\' to separate them from the rest of the characters. The argument may precede or follow the adjoining text, but there must be no intervening blanks between the concatenation operator and the rest of the characters. To position an argument between two alphanumeric characters, place a backslash both before and after the argument name. For example, consider the following macro definition:

```
SWAP_MEM  MACRO REG1,REG2       ;swap memory contents
          LD    A,[I\REG1]      ;using A as temp
          LD    B,[I\REG2]      ;using B as temp
          LD    [I\REG1],B
          LD    [I\REG2],A
          ENDM
```

If this macro were called with the following statement,

```
SWAP_MEM X,Y
```

then for the macro expansion, the macro processor would substitute the character 'X' for the dummy argument REG1, and the character 'Y' for the dummy argument REG2. The concatenation operator (\) indicates to the macro processor that the substitution characters for the dummy arguments are to be concatenated in both cases with the character I. The resulting expansion of this macro call would be:

```
LD    A,[IX]
LD    B,[IY]
LD    [IX],B
LD    [IY],A
```

2.5.5.2 Return Value Operator - ?

Another macro definition operator is the question mark (**?**) that returns the value of a symbol. When the macro processor encounters this operator, the **?***symbol* sequence is converted to a character string representing the decimal value of the *symbol*. For example, consider the following modification of the SWAP_MEM macro described above:

```
SWAP_MEM  MACRO REG1,REG2       ;swap memory contents
          LD    A,[_lab\?REG1]  ;using A as temp
          LD    B,[_lab\?REG2]  ;using B as temp
          LD    [_lab\?REG1],B
          LD    [_lab\?REG2],A
          ENDM
```

If the source file contained the following SET statements and macro call,

```
AREG      SET           1
BREG      SET           2
          SWAP_MEM      AREG,BREG
```

then the sequence of events would be as follows: the macro processor would first substitute the characters AREG for each occurrence of REG1 and BREG for each occurrence of REG2. For discussion purposes (this would never appear on the source listing), the intermediate macro expansion would be:

```
LD    A,[_lab\?AREG]
LD    B,[_lab\?BREG]
LD    [_lab\?AREG],B
LD    [_lab\?BREG],A
```

The macro processor would then replace **?**AREG with the character X and **?**BREG with the character Y, since X is the value of the symbol AREG and Y is the value of BREG. The resulting intermediate expansion would be:

```
        LD      A,[_lab\1]
        LD      B,[_lab\2]
        LD      [_lab\1],B
        LD      [_lab\2],A
```

Next, the macro processor would apply the concatenation operator (\), and the resulting expansion as it would appear on the source listing would be:

```
        LD      A,[_lab1]
        LD      B,[_lab2]
        LD      [_lab1],B
        LD      [_lab2],A
```

2.5.5.3 Return Hex Value Operator - %

The percent sign (**%**) is similar to the standard return value operator except that it returns the hexadecimal value of a symbol. When the macro processor encounters this operator, the **%***symbol* sequence is converted to a character string representing the hexadecimal value of the *symbol*. Consider the following macro definition:

```
GEN_LAB    MACRO   LAB,VAL,STMT
LAB\%VAL:  STMT
           ENDM
```

This macro generates a label consisting of the concatenation of the label prefix argument and a value that is interpreted as hexadecimal. If this macro were called as follows,

```
NUM     SET      10
        GEN_LAB  HEX,NUM,'NOP'
```

the macro processor would first substitute the characters HEX for LAB, then it would replace **%**VAL with the character A, since A is the hexadecimal representation for the decimal integer 10. Next, the macro processor would apply the concatenation operator (\). Finally, the string 'NOP' would be substituted for the STMT argument. The resulting expansion as it would appear in the listing file would be:

```
HEXA:   NOP
```

The percent sign is also the character used to indicate a binary constant. If a binary constant is required inside a macro it may be necessary to enclose the constant in parentheses or escape the constant by following the percent sign by a backslash (\).

2.5.5.4 Dummy Argument String Operator - "

Another dummy argument operator is the double quote ("). This character is replaced with a single quote by the macro processor, but following characters are still examined for dummy argument names. The effect in the macro call is to transform any enclosed dummy arguments into literal strings. For example, consider the following macro definition:

```
STR_MAC MACRO   STRING
        ASCII   "STRING"
        ENDM
```

If this macro were called with the following macro expansion line,

```
        STR_MAC ABCD
```

then the resulting macro expansion would be:

```
        ASCII   'ABCD'
```

Double quotes also make possible **DEFINE** directive expansion within quoted strings. Because of this overloading of the double quotes, care must be taken to insure against inappropriate expansions in macro definitions. Since **DEFINE** expansion occurs before macro substitution, any **DEFINE** symbols are replaced first within a macro dummy argument string:

```
        DEFINE  LONG  'short'
STR_MAC MACRO   STRING
        MSG     'This is a LONG STRING'
        MSG     "This is a LONG STRING"
        ENDM
```

If this macro were invoked as follows,

```
        STR_MAC sentence
```

then the resulting expansion would be:

```
        MSG     'This is a LONG STRING'
        MSG     'This is a short sentence'
```

2.5.5.5 Macro Local Label Operator - ^

It may be desirable to pass a name as a macro argument to be used as a local address reference within the macro body. If a circumflex (^) precedes an identifier then the macro preprocessor will perform name mangling on that label so the label is used literally in the resulting macro expansion. Here is an example:

```
LOAD    MACRO   ADDR
        LD      A,[^ADDR]
        ENDM
```

The macro ^-operator performs name mangling on the ADDR argument. Consider the following macro call:

```
_LOCAL: LOAD    _LOCAL
```

With the local label in the macro definition the macro LOAD would expand to the something like this:

```
_LOCAL:
        LD      A,[_LOCAL__M_L000001]
```

This would result in an assembly error as the label `LOCAL__M_L000001` is nowhere defined. Without the local label operator in the macro definition (as shown above) the macro LOAD would expand, as expected, to this:

```
_LOCAL:
        LD      A,[_LOCAL]
```

This will assemble correctly.

2.5.6 DUP, DUPA, DUPC, DUPF Directives

The **DUP**, **DUPA**, **DUPC**, and **DUPF** directives are specialized macro forms. They can be thought of as a simultaneous definition and call of an unnamed macro. The source statements between the **DUP**, **DUPA**, **DUPC**, and **DUPF** directives and the **ENDM** directive follow the same rules as macro definitions, including (in the case of **DUPA**, **DUPC**, and **DUPF**) the dummy operator characters described previously. For a detailed description of these directives, refer to Section 2.6, "Assembler Directives".

2.5.7 Conditional Assembly

Conditional assembly facilitates the writing of comprehensive source programs that can cover many conditions. Assembly conditions may be specified through the use of arguments in the case of macros, and through definition of symbols via the **DEFINE**, **SET**, and **EQU** directives. Variations of parameters can then cause assembly of only those parts necessary for the given conditions. The built-in functions of the assembler provide a versatile means of testing many conditions of the assembly environment (see Section 2.4.4, "Functions", for more information on the assembler built-in functions).

Conditional directives can also be used within a macro definition to ensure at expansion time that arguments fall within a range of allowable values. In this way macros become self-checking and can generate error messages to any desired level of detail.

The conditional assembly directive **IF** has the following form:

```
IF      expression
 .
 .
[ELSE]  ;(the ELSE directive is optional)
 .
 .
ENDIF
```

A section of a program that is to be conditionally assembled must be bounded by an **IF-ENDIF** directive pair. If the optional **ELSE** directive is not present, then the source statements following the **IF** directive and up to the next **ENDIF** directive will be included as part of the source file being assembled only if the *expression* had a nonzero result. If the *expression* has a value of zero, the source file will be assembled as if those statements between the **IF** and the **ENDIF** directives were never encountered. If the **ELSE** directive is present and *expression* has a nonzero result, then the statements between the **IF** and **ELSE** directives will be assembled, and the statement between the **ELSE** and **ENDIF** directives will be skipped. Alternatively, if *expression* has a value of zero, then the statements between the **IF** and **ELSE** directives will be skipped, and the statements between the **ELSE** and **ENDIF** directives will be assembled.

2.6 Assembler Directives

2.6.1 Overview

Assembler directives, or pseudo instructions, are used to control the assembly process. Rather than being translated into an S1C88 machine instruction, assembler directives are interpreted by the assembler. The directives perform actions such as assembly control, listing control, defining symbols or changing the location counter. Upper and lower case letters are considered equivalent for assembler directives.

Assembler directives can be grouped by function into five types:

1. Debugging
2. Assembly control
3. Symbol definition
4. Data definition/storage allocation
5. Macros and conditional assembly

2.6.1.1 Debugging

The compiler generates the following directives to pass high level language symbolic debug information via the assembler into the object file:

CALLS - Pass call information to object file. Used to build a call tree at link time for overlaying overlay sections.

SYMB - Pass symbolic debug information

2.6.1.2 Assembly Control

The directives used for assembly control are:

ALIGN - Specify alignment

COMMENT - Start comment lines. This directive is not permitted in IF/ELSE/ENDIF constructs and MACRO/DUP definitions.

DEFINE - Define substitution string

DEFSECT - Define section name and attributes

END - End of source program

FAIL - Programmer generated error message

INCLUDE - Include secondary file

MSG - Programmer generated message

RADIX - Change input radix for constants

SECT - Activate section

UNDEF - Undefine **DEFINE** symbol

WARN - Programmer generated warning

2.6.1.3 Symbol Definition

The directives used to control symbol definition are:

EQU - Equate symbol to a value;
accepts forward references

EXTERN - External symbol declaration;
also permitted in module body

GLOBAL - Global symbol declaration;
also permitted in module body

LOCAL - Local symbol declaration

NAME - Identify object file

SET - Set symbol to a value;
accepts forward references

2.6.1.4 Data Definition/Storage Allocation

The directives used to control constant data definition and storage allocation are:

ASCII - Define ASCII string

ASCIZ - Define NULL padded ASCII string

DB - Define constant byte

DS - Define storage

DW - Define constant word

2.6.1.5 Macros and Conditional Assembly

The directives used for macros and conditional assembly are:

DUP - Duplicate sequence of source lines

DUPA - Duplicate sequence with arguments

DUPC - Duplicate sequence with characters

DUPF - Duplicate sequence in loop

ENDIF - End of conditional assembly

ENDM - End of macro definition

EXITM - Exit macro

IF - Conditional assembly directive

MACRO - Macro definition

PMACRO - Purge macro definition

2.6.2 ALIGN Directive

Syntax:

ALIGN *expression*

Description:

Align the location counter. The *expression* must be represented by a value of 2^k. The default alignment is on a multiple of 1 byte. *expression* must be greater than 0. If *expression* is not a value of 2^k, a warning is issued and the alignment will be set to the next 2^k value. Alignment will be performed once at the place where you write the **align** pseudo. The start of a section is aligned automatically to the largest alignment value occurring in that section.

Depending on the section type the assembler has two cases for this directive.

- Relocatable sections
  The section will be aligned on the calculated alignment boundary. A gap is generated depending on the current relative location counter for this section.
- Absolute sections
  The section location is not changed.
  A gap is generated according to the current absolute address.

Examples:

```
ALIGN 4         ;align at 4 bytes
lab1: ALIGN 6   ;not a 2^k value.
                ;a warning is issued
                ;lab1 is aligned on 8 bytes
```

2.6.3 ASCII Directive

Syntax:

[*label*:] **ASCII** *string* [, *string*]...

Description:

Define list of ASCII characters. The **ASCII** directive allocates and initializes an array of memory for each *string* argument. No NULL byte is added to the end of the array. Therefore, the behavior is identical to the DB directive with a string argument.

See also:

ASCIZ, DB

Examples:

```
HELLO:  ASCII   "Hello world"   ;Is the same as  DB "Hello world"
```

2.6.4 ASCIZ Directive

Syntax:

[*label*:] **ASCIZ** *string* [, *string*]...

Description:

Define list of ASCII characters. The **ASCIZ** directive allocates and initializes an array of memory for each *string* argument. A NULL byte is added to the end of each array.

See also:

ASCII, DB

Examples:

```
HELLO:  ASCIZ   "Hello world"   ;Is the same as  DB "Hello world",0
```

2.6.5 CALLS Directive

Syntax:

CALLS *'caller', 'callee'*[, *nr*]... [, *'callee'*[, *nr*]...]...

Description:

Create a flow graph reference between *caller* and *callees*. The linker needs this information to build a flow graph, which steers the overlay algorithm. *caller* and *callee* are names of functions

Stack information is also specified. After the callee name, for each possible stack a usage count can be specified (e.g. system stack, user stack). The value specified (*nr*) is the stack usage (in bytes for the S1C88) at the time of the call including the 'RET' address of the current function. Currently the S1C88 tool only use the system stack.

This information is used by the linker to compute the used stack within the application. The information is found in the generated linker map file (**-M** option) within the call graph.

When *callee* is an empty name, this means we define the stack usage of the function itself.

See also Section 2.2.3.1, "Section Names", and Section 3.4, "Linker Output".

Examples:

```
DEFSECT "OVLN@nfunc", DATA, OVERLAY, SHORT
DEFSECT "OVLN@main", DATA, OVERLAY, SHORT
CALLS   'main', 'nfunc', 5
```

2.6.6 COMMENT Directive

Syntax:

COMMENT *delimiter*
:
delimiter

Description:

Start Comment Lines. The **COMMENT** directive is used to define one or more lines as comments. The first non-blank character after the **COMMENT** directive is the comment delimiter. The two delimiters are used to define the comment text. The line containing the second comment delimiter will be considered the last line of the comment. The comment text can include any printable characters and the comment text will be produced in the source listing as it appears in the source file.

A label is not allowed with this directive.

This directive is not permitted in IF/ELSE/ENDIF constructs and MACRO/DUP definitions.

Examples:

```
COMMENT         + This is a one line comment +
COMMENT         *     This is a multiple line comment. Any number of lines
                      can be placed between the two delimiters.
                *
```

2.6.7 DB Directive

Syntax:

[*label*:] **DB** *arg*[, *arg*]...

Description:

Define Constant Byte. The **DB** directive allocates and initializes a byte of memory for each *arg* argument. *arg* may be a numeric constant, a single or multiple character string constant, a symbol, or an expression. The **DB** directive may have one or more arguments separated by commas. Multiple arguments are stored in successive address locations. If multiple arguments are present, one or more of them can be null (two adjacent commas), in which case the corresponding address location will be filled with zeros. An error will occur if the evaluated argument value is too large to represent in a single byte.
label, if present, will be assigned the value of the runtime location counter at the start of the directive processing.
Integer arguments are stored as is, but must be byte values (e.g. within the range 0–255). Single and multiple character strings are handled in the following manner:

1. Single character strings are stored in a byte whose bits represent the ASCII value of the character.
   Example: `'R' = 52H`
2. Multiple character strings represent bytes composed of the ASCII representation of the characters in the string.
   Example: `'ABCD' = 41H, 42H, 43H, 44H`

See also:

DS, DW

Examples:

```
TABLE:  DB      14,253,62H,'ABCD'
CHARS:  DB      'A','B','C','D'
```

2.6.8 DEFINE Directive

Syntax:

DEFINE *symbol* *string*

Description:

Define Substitution String. The **DEFINE** directive is used to define substitution strings that will be used on all following source lines. All succeeding lines will be searched for an occurrence of *symbol*, which will be replaced by *string*. This directive is useful for providing better documentation in the source program. *symbol* must adhere to the restrictions for labels. That is, the first character must be alphabetic or the underscore (_), and the remainder of which must be either alphanumeric or the underscore (_). A warning will result if a new definition of a previously defined symbol is attempted. Macros represent a special case. **DEFINE** directive translations will be applied to the macro definition as it is encountered. When the macro is expanded any active **DEFINE** directive translations will again be applied.
A label is not allowed with this directive.

See also:

UNDEF

Examples:

If the following **DEFINE** directive occurred in the first part of the source program:

```
        DEFINE  ARRAYSIZ        '10 * SAMPLSIZ'
```

then the source line below:

```
        DS      ARRAYSIZ
```

would be transformed by the assembler to the following:

```
        DS      10 * SAMPLSIZ
```

2.6.9 DEFSECT Directive

Syntax:

DEFSECT *section*, *type* [, *attr*]... [**AT** *address*]

Description:

Use this directive to define section names and declaration attributes. Before any code or data can be placed in a section, you must use the SECT directive to activate the section. The definition can have declaration attributes and must have a section type (*type*).

The section type can be:

type: **DATA** | **CODE**

The section declaration attribute can be:

attr:

Group1: **SHORT** | **TINY**
Group2: **FIT 100H** | **FIT 8000H** | **FIT 10000H**
Group3: **OVERLAY** | **ROMDATA** | **NOCLEAR** | **CLEAR** | **INIT** | **MAX**
Group4: **JOIN**

For each group one attribute can be specified at the most. **CLEAR** sections are zeroed at startup. This attribute can only be used on a DATA type section.

Sections with the **NOCLEAR** attribute are not zeroed at startup. This is a default attribute for DATA sections. The attribute can only be used for DATA sections.

The **INIT** attribute defines that the DATA section contains initialization data, which is copied from ROM to RAM at program startup.

A section becomes overlayable by specifying the **OVERLAY** attribute. Only DATA sections are overlayable.

ROMDATA sections (allowed on DATA and CODE sections) contain data to be placed in ROM. This ROM area is not executable.

When DATA sections with the same name occur in different object modules with the **MAX** attribute, the linker generates a section of which the size is the maximum of the sizes in the individual object modules. The **MAX** attribute only applies to DATA sections.

The **SHORT** attribute specifies for CODE sections that the section must be located within the first 32K (common, non-banked area). When used on a DATA section, the **SHORT** attribute specifies that the section has to be located within the first 64K page.

The **TINY** attribute specifies that the section must be located within one page of 256 bytes maximum, within the first 64K of data memory.

The **FIT** attributes specify that a section may not cross a given boundary. As a result, the specified size is also the maximum possible size for such a section (be aware, the linker links all sections with the same name together and the check will be done on the resulting section). So, for example a `FIT 8000H` section may be located within range `0` and `7FFFH` or within `8000H` and `0FFFFH`. It cannot be positioned across address `8000H` or `10000H`, etc.

You can group sections together in one page with the **JOIN** attribute. The **JOIN** attribute should always be used together with the **FIT** attribute. For example, when more sections have to be located within the same data page, you can use this attribute. See also Section 2.2.3.3, "Grouped Sections".

See Section 2.2.3.1, "Section Names", for detailed information about sections, section types and section attributes.

See also:

SECT

Examples:

```
DEFSECT ".text", DATA   ;declare section .text
SECT    ".text"         ;switch to section .text
```

2.6.10 DS Directive

Syntax:

[*label*:] **DS** *expression*

Description:

Define Storage. The **DS** directive reserves a block of memory the length of which in bytes is equal to the value of *expression*. This directive causes the runtime location counter to be advanced by the value of the absolute integer expression in the operand field. The block of memory reserved is not initialized to any value. The expression must be an integer greater than zero and cannot contain any forward references to address labels (labels that have not yet been defined).

label, if present, will be assigned the value of the runtime location counter at the start of the directive processing.

See also:

DB, DW

Examples:

```
S_BUF:  DS      12      ; Sample buffer
```

2.6.11 DUP Directive

Syntax:

[*label*:] **DUP** *expression*
:
ENDM

Description:

Duplicate Sequence of Source Lines. The sequence of source lines between the **DUP** and **ENDM** directives will be duplicated by the number specified by the integer *expression*. If the expression evaluates to a number less than or equal to 0, the sequence of lines will not be included in the assembler output. The expression result must be an absolute integer and cannot contain any forward references to address labels (labels that have not already been defined). The **DUP** directive may be nested to any level.

label, if present, will be assigned the value of the runtime location counter at the start of the **DUP** directive processing.

See also:

DUPA, DUPC, DUPF, ENDM, MACRO

Examples:

The sequence of source input statements,

```
COUNT   SET     3
        DUP     COUNT   ; SRA BY COUNT
        SRA     A
        ENDM
```

would generate the following in the source listing:

```
COUNT   SET     3
        DUP     COUNT   ; SRA BY COUNT
        SRA     A
        ENDM

;       SRA     A
;       SRA     A
;       SRA     A
```

2.6.12 DUPA Directive

Syntax:

[*label*:] **DUPA** *dummy*, *arg*[, *arg*]...
:
ENDM

Description:

Duplicate Sequence With Arguments. The block of source statements defined by the **DUPA** and **ENDM** directives will be repeated for each argument. For each repetition, every occurrence of the dummy parameter within the block is replaced with each succeeding argument string. If the argument string is a null, then the block is repeated with each occurrence of the dummy parameter removed. If an argument includes an embedded blank or other assembler-significant character, it must be enclosed with single quotes.

label, if present, will be assigned the value of the runtime location counter at the start of the **DUPA** directive processing.

See also:

DUP, DUPC, DUPF, ENDM, MACRO

Examples:

If the input source file contained the following statements,

```
        DUPA    VALUE,12,32,34
        DB      VALUE
        ENDM
```

then the assembler source listing would show

```
        DUPA    VALUE,12,32,34
        DB      VALUE
        ENDM
;       DB      12
;       DB      32
;       DB      34
```

2.6.13 DUPC Directive

Syntax:

[*label*:] **DUPC** *dummy*, *string*
:
ENDM

Description:

Duplicate Sequence With Characters. The block of source statements defined by the **DUPC** and **ENDM** directives will be repeated for each character of *string*. For each repetition, every occurrence of the dummy parameter within the block is replaced with each succeeding character in the string. If the string is null, then the block is skipped.

label, if present, will be assigned the value of the runtime location counter at the start of the **DUPC** directive processing.

See also:

DUP, DUPA, DUPF, ENDM, MACRO

Examples:

If the input source file contained the following statements,

```
        DUPC    VALUE,'123'
        DB      VALUE
        ENDM
```

then the assembler source listing would show

```
        DUPC    VALUE,'123'
        DB      VALUE
        ENDM
;       DB      1
;       DB      2
;       DB      3
```

2.6.14 DUPF Directive

Syntax:

[*label*:] **DUPF** *dummy*, [*start*], *end*[, *increment*]
.
.
ENDM

Description:

Duplicate Sequence In Loop. The block of source statements defined by the **DUPF** and **ENDM** directives will be repeated in general (*end* - *start*) + 1 times when *increment* is 1. *start* is the starting value for the loop index; *end* represents the final value. *increment* is the increment for the loop index; it defaults to 1 if omitted (as does the *start* value). The *dummy* parameter holds the loop index value and may be used within the body of instructions.

label, if present, will be assigned the value of the runtime location counter at the start of the **DUPF** directive processing.

See also:

DUP, DUPA, DUPC, ENDM, MACRO

Examples:

If the input source file contained the following statements,

```
        DUPF    NUM,0,3
        LD      [NUM],A
        ENDM
```

then the assembler source listing would show

```
        DUPF    NUM,0,3
        LD      [NUM],A
        ENDM
;       LD      [0],A
;       LD      [1],A
;       LD      [2],A
;       LD      [3],A
```

2.6.15 DW Directive

Syntax:

[*label*:] **DW** *arg*[, *arg*]...

Description:

Define Constant Word. The **DW** directive allocates and initializes a word of memory for each *arg* argument. *arg* may be a numeric constant, a single or double character string constant, a symbol, or an expression. The **DW** directive may have one or more arguments separated by commas. Multiple arguments are stored in successive address locations. If multiple arguments are present, one or more of them can be null (two adjacent commas), in which case the corresponding address location will be filled with zeros. An error will occur if the evaluated argument value is too large to represent in a single word.

label, if present, will be assigned the value of the runtime location counter at the start of the directive processing.

Note that word values are stored in memory with the lower 8 bits on the lowest address.

Integer arguments are stored as is. Single and multiple character strings are handled in the following manner:

1. Single character strings are stored in a word whose lower seven bits represent the ASCII value of the character.

   Example: `'R' = 52H`

2. Multiple character strings consisting of more than two characters are not allowed. Two-character strings are stored as if the ASCII value of the first character is the high byte value of the word. The second character is used as the low byte.

   Example: `'AB' = 4142H`

See also:

DB, DS

Examples:

```
TABLE:  DW      14,1635,2662H,'AB'
```

is equal to

```
TABLE:  DB      14,0,1635%256,6,62H,26H,'B','A'
```

2.6.16 END Directive

Syntax:

END

Description:

End of Source Program. The optional **END** directive indicates that the logical end of the source program has been encountered. The **END** directive cannot be used in a macro expansion.

A label is not allowed with this directive.

Examples:

```
END             ;End of source program
```

2.6.17 ENDIF Directive

Syntax:

ENDIF

Description:

End of Conditional Assembly. The **ENDIF** directive is used to signify the end of the current level of conditional assembly. Conditional assembly directives can be nested to any level, but the **ENDIF** directive always refers to the most previous **IF** directive. A label is not allowed with this directive.

See also:

IF

Examples:

```
IF      DEB             ;Report building of the debug version
MSG     'Debug Version'
ENDIF
```

2.6.18 ENDM Directive

Syntax:

ENDM

Description:

End of Macro Definition. Every **MACRO**, **DUP**, **DUPA**, and **DUPC** directive must be terminated by an **ENDM** directive. A label is not allowed with this directive.

See also:

DUP, DUPA, DUPC, MACRO

Examples:

```
SWAP_MEM  MACRO REG1,REG2       ;swap memory contents
          LD    A,[I\REG1]      ;using A as temp
          LD    B,[I\REG2]      ;using B as temp
          LD    [I\REG1],B
          LD    [I\REG2],A
          ENDM
```

2.6.19 EQU Directive

Syntax:

name **EQU** *expression*

Description:

Equate Symbol to a Value. The **EQU** directive assigns the value of *expression* to the symbol *name*. The **EQU** directive is one of the directives that assigns a value other than the program counter to the name. The symbol name cannot be redefined anywhere else in the program. The *expression* may be relative or absolute, and forward references are allowed.
An **EQU** symbol can be made global.

See also:

SET

Examples:

```
A_D_PORT  EQU   4000H
```

This would assign the value 4000H to the symbol A_D_PORT.

2.6.20 EXITM Directive

Syntax:

EXITM

Description:

Exit Macro. The **EXITM** directive will cause immediate termination of a macro expansion. It is useful when used with the conditional assembly directive **IF** to terminate macro expansion when error conditions are detected.
A label is not allowed with this directive.

See also:

DUP, DUPA, DUPC, MACRO

Examples:

```
CALC    MACRO   XVAL,YVAL
        IF      XVAL<0
        MSG     'Macro parameter value out of range'
        EXITM   ;Exit macro
        ENDIF
         :
        ENDM
```

2.6.21 EXTERN Directive

Syntax:

EXTERN [(*attrib*[, *attrib*]...)] *symbol*[, *symbol*]...

Description:

External Symbol Declaration. The **EXTERN** directive is used to specify that the list of symbols is referenced in the current module, but is not defined within the current module. These symbols must either have been defined outside of any module or declared as globally accessible within another module using the **GLOBAL** directive.

The optional argument *attrib* can be one of the following symbol attributes:

CODE symbol is in ROM

DATA symbol is in RAM

SHORT symbol is within first page of memory
for CODE in first 32K
for DATA in first 64K

TINY symbol is in one page of 256 bytes maximum of the first 64K page of DATA

If the **EXTERN** directive is not used to specify that a symbol is defined externally and the symbol is not defined within the current module, a warning is generated, and an **EXTERN** symbol is inserted.

A label is not allowed with this directive.

See also:

GLOBAL

Examples:

```
EXTERN  AA,CC,DD                ;defined elsewhere
EXTERN  (CODE,SHORT) EE         ;within first 32K of code memory
```

2.6.22 FAIL Directive

Syntax:

FAIL [{*str* | *exp*} [, {*str* | *exp*}]...]

Description:

Programmer Generated Error. The **FAIL** directive will cause an error message to be output by the assembler. The total error count will be incremented as with any other error. The **FAIL** directive is normally used in conjunction with conditional assembly directives for exceptional condition checking. The assembly stops immediately after the error has been printed. An arbitrary number or strings and expressions, in any order but separated by commas, can be specified optionally to describe the nature of the generated error.

A label is not allowed with this directive.

See also:

MSG, WARN

Examples:

```
FAIL    'Parameter out of range'
```

2.6.23 GLOBAL Directive

Syntax:

GLOBAL *symbol*[,*symbol*]...

Description:

Global Section Symbol Declaration. The **GLOBAL** directive is used to specify that the list of symbols is defined within the current section or module, and that those definitions should be accessible by all modules. If the symbols that appear in the operand field are not defined in the module, an error will be generated. Symbols that are defined "global" are accessible from other modules using the **EXTERN** directive.

A label is not allowed with this directive.

Only program labels and **EQU** labels can be made global.

See also:

EXTERN, LOCAL

Examples:

```
GLOBAL  LOOPA   ;LOOPA will be globally
                ;accessible by other modules
```

2.6.24 IF Directive

Syntax:

IF *expression*
:
[**ELSE**] (the **ELSE** directive is optional)
:
ENDIF

Description:

Conditional Assembly Directive. Part of a program that is to be conditionally assembled must be bounded by an **IF-ENDIF** directive pair. If the optional **ELSE** directive is not present, then the source statements following the **IF** directive and up to the next **ENDIF** directive will be included as part of the source file being assembled only if the *expression* has a nonzero result. If the *expression* has a value of zero, the source file will be assembled as if those statements between the **IF** and the **ENDIF** directives were never encountered. If the **ELSE** directive is present and *expression* has a nonzero result, then the statements between the **IF** and **ELSE** directives will be assembled, and the statements between the **ELSE** and **ENDIF** directives will be skipped. Alternatively, if *expression* has a value of zero, then the statements between the **IF** and **ELSE** directives will be skipped, and the statements between the **ELSE** and **ENDIF** directives will be assembled.

The *expression* must have an absolute integer result and is considered true if it has a nonzero result. The *expression* is false only if it has a result of 0. Because of the nature of the directive, *expression* must be known on pass one (no forward references allowed). **IF** directives can be nested to any level. The **ELSE** directive will always refer to the nearest previous **IF** directive as will the **ENDIF** directive.

A label is not allowed with this directive.

See also:

ENDIF

Examples:

```
IF      XVAL<0
MSG     'Please select larger value for XVAL'
ENDIF
```

2.6.25 INCLUDE Directive

Syntax:

INCLUDE *string* | *<string>*

Description:

Include Secondary File. This directive is inserted into the source program at any point where a secondary file is to be included in the source input stream. The string specifies the filename of the secondary file. The filename must be compatible with the operating system and can include a directory specification.

The file is searched for first in the current directory, unless the *<string>* syntax is used, or in the directory specified in *string*. If the file is not found, the assembler searches in a directory specified with this environment variable AS88INC. AS88INC contain more than one directory. Finally, the directory `..\include` relative to the directory where the assembler binary is located is searched. If the *<string>* syntax is given, the directory specified in *string* and the current directory are not searched. However, the directories specified with AS88INC and the relative path are still searched.

A label is not allowed with this directive.

Examples:

```
INCLUDE 'storage\mem.asm'
INCLUDE <data.asm>              ; Do not look in current directory
```

2.6.26 LOCAL Directive

Syntax:

LOCAL *symbol*[, *symbol*]...

Description:

Local Section Symbol Declaration. The **LOCAL** directive is used to specify that the list of symbols is defined within the current module, and that those definitions are explicitly local to that section or module. It is useful in cases where a symbol may not be exported outside of the module (as labels in a module are defined "global" by default). If the symbols that appear in the operand field are not defined in the module, an error will be generated.

A label is not allowed with this directive.

See also:

GLOBAL

Examples:

```
LOCAL   LOOPA   ;LOOPA local to this module
```

2.6.27 MACRO Directive

Syntax:

name **MACRO** [*dummy argument list*]

:

macro definition statements

:

ENDM

Description:

Macro Definition. The dummy argument list has the form:

[*dumarg*[, *dumarg*]...]

The required name is the symbol by which the macro will be called.

The definition of a macro consists of three parts: the header, which assigns a name to the macro and defines the dummy arguments; the body, which consists of prototype or skeleton source statements; and the terminator. The header is the **MACRO** directive, its name, and the dummy argument list. The body contains the pattern of standard source statements. The terminator is the **ENDM** directive.

The dummy arguments are symbolic names that the macro processor will replace with arguments when the macro is expanded (called). Each dummy argument must obey the same rules as symbol names. Within each of the three dummy argument field, the dummy arguments are separated by commas. The dummy argument fields are separated by one or more blanks.

Macro definitions may be nested but the nested macro will not be defined until the primary macro is expanded.

Section 2.5, "Macro Operations", contains a complete description of macros.

See also:

DUP, DUPA, DUPC, DUPF, ENDM

Examples:

```
SWAP_MEM MACRO REG1,REG2       ;swap memory contents
         LD    A,[I\REG1]      ;using A as temp
         LD    B,[I\REG2]      ;using B as temp
         LD    [I\REG1],B
         LD    [I\REG2],A
         ENDM
```

2.6.28 MSG Directive

Syntax:

MSG [{*str* | *exp*}[, {*str* | *exp*}]...]

Description:

Programmer Generated Message. The **MSG** directive will cause a message to be output by the assembler. The error and warning counts will not be affected. The **MSG** directive is normally used in conjunction with conditional assembly directives for informational purposes. The assembly proceeds normally after the message has been printed. An arbitrary number of strings and expressions, in any order but separated by commas, can be specified optionally to describe the nature of the message.

A label is not allowed with this directive.

See also:

FAIL, WARN

Examples:

```
MSG     'Generating tables'
```

2.6.29 NAME Directive

Syntax:

NAME "*str*"

Description:

The **NAME** directive is used by the assembler to give an identification to the produced object file. The linker and locator can then use this information to identify the source within the map files. Also a debugger may display the value as a 'module' name.

When this directive is omitted, the assembler will use the module's source name as an identification. When using the control program, this name might become a 'random' name.

Examples:

```
NAME  "strcat"  ;object is identified by the name "strcat"
```

2.6.30 PMACRO Directive

Syntax:

PMACRO *symbol*[, *symbol*]...

Description:

Purge Macro Definition. The specified macro definition will be purged from the macro table, allowing the macro table space to be reclaimed.

A label is not allowed with this directive.

See also:

MACRO

Examples:

```
PMACRO  MAC1,MAC2
```

This statement would cause the macros named MAC1 and MAC2 to be purged.

2.6.31 RADIX Directive

Syntax:

RADIX *expression*

Description:

Change Input Radix for Constants. Changes the input base of constants to the result of *expression*. The absolute integer expression must evaluate to one of the legal constant bases (2, 8, 10, or 16). The default radix is 10. The **RADIX** directive allows the programmer to specify constants in a preferred radix without a leading radix indicator. The radix suffix for base 10 numbers is the 'D' character. Note that if a constant is used to alter the radix, it must be in the appropriate input base at the time the **RADIX** directive is encountered.

A label is not allowed with this directive.

Examples:

```
_RAD10: DB      10      ; Evaluates to hex A
        RADIX   2
_RAD2:  DB      10      ; Evaluates to hex 2
        RADIX   16D
_RAD16: DB      10      ; Evaluates to hex 10
        RADIX   3       ; Bad radix expression
```

2.6.32 SECT Directive

Syntax:

SECT "*str*" [, **RESET**]

Description:

The **SECT** directive flags the assembler that another section, with name *str*, becomes active. Before a section can be activated for the first time, it must be defined first, by the **DEFSECT** directive. Subsequent activations can be done by the **SECT** directive only.

You can use the section attribute RESET to reset counting storage allocation in DATA sections with section attribute MAX.

See Section 2.2.3.1, "Section Names", for detailed information about sections.

See also:

DEFSECT

Examples:

```
DEFSECT  ".text", DATA ;declare section .text
SECT     ".text"       ;switch to section .text
```

2.6.33 SET Directive

Syntax:

name **SET** *expression*

Description:

Set Symbol to a Value. The **SET** directive is used to assign the value of the expression in the operand field to the symbol name. The **SET** directive functions somewhat like the **EQU** directive. However, symbols defined via the **SET** directive can have their values redefined in another part of the program (but only through the use of another **SET** directive). The **SET** directive is useful in establishing temporary or reusable counters within macros. The expression in the operand field of a **SET** may have forward references.

SET symbols cannot be made global.

See also:

EQU

Examples:

```
COUNT  SET      0       ; Initialize COUNT
```

2.6.34 SYMB Directive

Syntax:

SYMB *string, expression*[*, abs_expr*] [*, abs_expr*]

Description:

The **SYMB** directive is used for passing high-level language symbolic debug information via the assembler (and linker/locator) to the debugger. *expression* can be any expression. *abs_expr* can be any expression resulting in an absolute value.

The **SYMB** directive is not meant for 'hand coded' assembly files. It is documented for completeness only and is supposed to be 'internal' to the tool chain.

2.6.35 UNDEF Directive

Syntax:

UNDEF *symbol*

Description:

Undefine **DEFINE** Symbol. The **UNDEF** directive causes the substitution string associated with *symbol* to be released, and *symbol* will no longer represent a valid **DEFINE** substitution. See the **DEFINE** directive for more information.

A label is not allowed with this directive.

See also:

DEFINE

Examples:

```
UNDEF   DEBUG    ;Undefines the DEBUG substitution string
```

2.6.36 WARN Directive

Syntax:

WARN [{*str* | *exp*}[, {*str* | *exp*}]...]

Description:

Programmer Generated Warning. The **WARN** directive will cause a warning message to be output by the assembler. The total warning count will be incremented as with any other warning. The **WARN** directive is normally used in conjunction with conditional assembly directives for exceptional condition checking. The assembly proceeds normally after the warning has been printed. An arbitrary number of strings and expressions, in any order but separated by commas, can be specified optionally to describe the nature of the generated warning.

A label is not allowed with this directive.

See also:

FAIL, MSG

Examples:

```
WARN    'parameter too large'
```

2.7 Assembler Controls

2.7.1 Introduction

Assembler controls are provided to alter the default behavior of the assembler. They can be specified on 'control lines', embedded in the source file. A control line is a line starting with a dollar sign (**$**). Such a line is not processed like a normal assembly source line, but as an assembler control line. One control per source line is allowed. An assembler control line may contain comments. Upper and lower case letters are considered equivalent for assembler directives.

The controls are classified as: primary or general.

Primary controls affect the overall behavior of the assembler and remain in effect throughout the assembly. For this reason, primary controls may only be used at the beginning of a source file, before the assembly starts. If you specify a primary control more than once, a warning message is given and the last definition is used. This enables you to override primary controls via command line options.

General controls are used to control the assembler during assembly. Control lines containing general controls may appear anywhere in a source file. When you specify general controls via the invocation line the corresponding general controls in the source file are ignored.

On the next pages, the available assembler controls are listed in alphabetic order. Some controls are set by default, and some controls have a default value.

2.7.2 Overview Assembler Controls

Table 2.7.2.1 Assembler controls

Control	Type	Default	Description
$CASE ON $CASE OFF	pri	ON	All user names are case sensitive. User names are not case sensitive.
$IDENT LOCAL $IDENT GLOBAL	pri	LOCAL	Default local labels. Default global labels.
$LIST ON $LIST OFF	gen		Resume listing. Stop listing.
$LIST "*flags*"	pri	cDEGlMnPQsWXy	Define what to include in/exclude from the list file.
$MODEL [S|C|D|L]	pri	L	Select memory model. Object files in different models cannot be linked together.
$STITLE "*title*"	gen		Set list page header title for next pages.
$TITLE "*title*"	pri	spaces	Set list page header title for first page.
$WARNING OFF $WARNING OFF *num*	pri		Suppress all warnings. Suppress one warning.
Type: Type of control: pri for primary controls, gen for general controls.			

2.7.3 Description of Assembler Controls

2.7.3.1 CASE

Control:

$CASE ON
$CASE OFF

Related option:

-c Set case sensitivity off; overrules the control.

Class:

Primary

Default:

$CASE ON

Description:

Selects whether the assembler operates in case sensitive mode or not. In case insensitive mode the assembler maps characters on input to uppercase (literal strings excluded).

Example:

```
;Begin of source
$case off       ;assembler in case insensitive mode
```

2.7.3.2 IDENT

Control:

$IDENT LOCAL
$IDENT GLOBAL

Related option:

-i[l | g] Default labels are local or global.

Class:

Primary

Default:

$IDENT LOCAL

Description:

With the $IDENT control you specify how a label is to be treated by the assembler. This is for code and data labels only. $IDENT LOCAL specifies that labels are local by default, with $IDENT GLOBAL labels are global by default.

SET identifiers are always treated as local symbols.

You can always overrule the default settings with the LOCAL or GLOBAL directives for a specific label.

Example:

```
;Begin of source
$ident global   ; assembly labels are global by default
```

2.7.3.3 LIST ON/OFF

Control:

$LIST ON
$LIST OFF

Related option:

-l Produce an assembler list file

Class:

General

Default:

$LIST ON

Description:

Switch the listing generation on or off. These controls take effect starting at the next line. Actual list file generation is selected on the command line. Without the command line option **-l**, no list file is produced.

Example:

```
$list off       ; Turn listing off.
                ; These lines are not present in the list file
   :
$list on        ; Turn listing back on.
                ; These lines are present in the list file
   :
```

2.7.3.4 LIST

Control:

$LIST "*flags*"

Related option:

-L[*flag*...] Remove specified source lines from list file

Class:

Primary

Default:

$LIST "cDEGlMnPQsWXy"

Description:

Specify which source lines are to be removed from the list file. The flags defined within the string are the same as for the **-L** command line option. See the **-L** option for an explanation of each flag available.

Example:

```
;Begin of source
$list "cw"      ; Remove source lines with assembler controls from the
                ; resulting list file and remove wrapped source lines
   :
```

2.7.3.5 MODEL

Control:

$MODEL [S | C | D | L]

Related option:

-M*model* Specify memory model

Class:

Primary

Default:

$MODEL L

Description:

With the $MODEL control you specify how the source must use the processor. You can specify four models:

Model	Description
S	small model, maximum of 64K code and data
C	compact code model, maximum of 64K code and 16M data
D	compact data model, maximum of 8M code and 64K data
L	large model, maximum of 8M code and 16M data

This means, for example, that in the small model, you should never change the CB/NB register value in the source, and also the EP/XP/YP registers must be fixed.

You cannot link object files that have been assembled for different models. This is to make sure the different models use the same approach to the page registers. The $MODEL control is used by the C compiler this way, but an assembler programmer is still able to select the 'wrong' model within a source. Thus writing a non-working program.

Example:

```
;Begin of source
$model s
; assemble using the small model
```

2.7.3.6 STITLE

Control:

$STITLE "*title*"

Related option:

-l Produce an assembler list file

Class:

General

Default:

$STITLE ""

Description:

Initialize Program Sub-Title. The $STITLE control initializes the program subtitle to the *title* in the operand field. The subtitle will be printed on the top of all succeeding pages until another $STITLE control is encountered. The subtitle is initially blank. The $STITLE control will not be printed in the source listing. An $STITLE control with no string argument will cause the current subtitle to be blank.

If the page width is too small for the title to fit in the header, it will be truncated.

See also:

TITLE

Example:

```
$stitle "Demo title"
; title in page header on succeeding pages
; is Demo title
```

2.7.3.7 TITLE

Control:

$TITLE "*title*"

Related option:

-l Produce an assembler list file

Class:

Primary

Default:

spaces

Description:

This control specifies the *title* to be used in the page heading of the first page of the list file.

If the page width is too small for the title to fit in the header, it will be truncated.

See also:

STITLE

Example:

```
;Begin of source
$title "NEWTITLE"
; title in page header on first page is NEWTITLE
```

2.7.3.8 WARNING

Control:

$WARNING OFF
$WARNING OFF *num*

Related option:

-w[*num*] Suppress one or all warning messages

Class:

Primary

Default:

– (All warnings enabled)

Description:

$WARNING suppresses all warnings. This is the same as **-w**. $WARNING OFF *num* suppresses one warning message, where *num* is the warning message number (same as the **-w***num* option).

Example:

```
;Begin of source
$warning off    ; switch all warnings off
```

CHAPTER 3 LINKER

3.1 Overview

This section gives a global overview of the process of linking programs for the S1C88 and its derivatives. The linker executable name for the S1C88 is **lk88**.

The linker combines relocatable object files, generated by the assembler, into one new relocatable object file (preferred extension `.out`). This file may be used as input in subsequent linker calls: the linkage process may be incremental. Normally the linker complains about unresolved external references. With incremental linking it is normal to have unresolved references in the output file. Incremental linking must be selected separately.

The linker can read normal object files and libraries of object modules. Modules in a library are included only when they are referenced. At the end of the linkage process the generated object, without unresolved references, will be called: a load module.

The S1C88 linker is an overlaying linker. The compiler generates overlayable sections. An overlayable section contains space reservations for variables which, at C level, are local to a function. If functions do not call each other, their local variables can be overlayed in memory. It is a task of the linker to combine function call information into a call graph and to determine upon the structure of this call graph how sections can be overlayed, using the smallest amount of RAM.

Incremental linkage disables overlaying, so the last link phase should not be incremental, even if the incremental phase resolves all externals.

The following diagram shows the input files and output files of the linker:

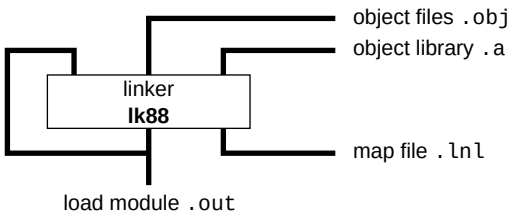


Fig. 3.1.1 S1C88 Linker

3.2 Linker Invocation

The invocation of the S1C88 linker is:

lk88 [*option*]... *file* ...

Options may appear in any order. Options start with a '**-**'. Only the **-lx** option is position dependent. Option may be combined: **-rM** is equal to **-r -M**. Options that require a filename or a string may be separated by a space or not: **-o***name* is equal to **-o** *name*.

file can be any object file (`.obj`), object libraries (`.a`) or incremental linker (`.out`) files. The files are linked in the same order as they appear on the command line.

The linker recognizes the following options:

Options Summary

Option	Description
-C	Link case insensitive (default case sensitive)
-L *directory*	Additional search path for system libraries
-L	Skip system library search
-M	Produce a link map (`.lnl`)
-N	Turn off overlaying
-O *name*	Specify basename of the resulting map files
-V	Display version header only
-c	Produce a separate call graph file (`.cal`)
-e	Clean up if erroneous result
-err	Redirect error messages to error file (`.elk`)
-f *file*	Read command line information from *file*, '-' means `stdin`
-l *x*	Search also in system library `lib`*x*`.a`
-o *filename*	Specify name of output file
-r	Suppress undefined symbol diagnostics
-u *symbol*	Enter *symbol* as undefined in the symbol table
-v or **-t**	Verbose option. Print name of each file as it is processed
-w *n*	Suppress messages above warning level *n*.

3.2.1 Detailed Description of Linker Options

-C

With this option **lk88** links case insensitive. The default is case sensitive linking.

-L [*directory*]

Add *directory* to the list of directories that are searched for system libraries. Directories specified with **-L** are searched before the standard directories specified by the environment variable C88LIB. If you specify **-L** without a directory, the environment variable C88LIB is not searched for system libraries. You may use the **-L** option more than once to add several directories to the search path for system libraries. The search path is created in the same order as in which the directories are specified on the command line.

Note: Directory names that include "O" (capital letter) cannot be specified with the **-L** option.

-M

Produce a link map (`.lnl`).

-N

Turn off overlaying. This can be useful for debugging.

-O *name*

Use *name* as the default basename for the resulting map files.

-V

With this option you can display the version header of the linker. This option must be the only argument of **lk88**. Other options are ignored. The linker exits after displaying the version header.

-c

Generate separate call graph file (`.cal`).

-e

Remove all link products such as temporary files, the resulting output file and the map file, in case an error occurred.

-err

The linker redirects error messages to a file with the same basename as the output file and the extension `.elk`. The default filename is `a.elk`.

-f *file*

Read command line information from *file*. If *file* is a '**-**', the information is read from standard input. You need to provide the EOF code to close `stdin`.

Use *file* for command line processing. To get around the limits on the size of the command line, it is possible to use command files. These command files contain the options that could not be part of the real command line. Command files can also be generated on the fly, for example by the make utility. More than one **-f** option is allowed.

Some simple rules apply to the format of the command file:

1. It is possible to have multiple arguments on the same line in the command file.
2. To include whitespace in the argument, surround the argument with either single or double quotes.
3. If single or double quotes are to be used inside a quoted argument, we have to go by the following rules:
   a. If the embedded quotes are only single or double quotes, use the opposite quote around the argument. Thus, if a argument should contain a double quote, surround the argument with single quotes.
   b. If both types of quotes are used, we have to split the argument in such a way that each embedded quote is surrounded by the opposite type of quote.

   Example:
   ```
   "This has a single quote ' embedded"
   ```
   or
   ```
   'This has a double quote " embedded'
   ```
   or
   ```
   'This has a double quote " and a single quote '"' embedded"
   ```
4. Some operating systems impose limits on the length of lines within a text file. To circumvent this limitation it is possible to use continuation lines. These lines end with a backslash and newline. In a quoted argument, continuation lines will be appended without stripping any whitespace on the next line. For non-quoted arguments, all whitespace on the next line will be stripped.

   Example:
   ```
   "This is a continuation \
   line"
           → "This is a continuation line"

   control(file1(mode,type),\
   file2(type))
           → control(file1(mode,type),file2(type))
   ```
5. It is possible to nest command line files up to 25 levels.

-l *x*

Search also in system library `lib`*x*`.a`, where *x* is a string. The linker first searches for system libraries in any directories specified with **-L***directory*, then in the standard directories specified with the environment variable C88LIB, unless the **-L** option is used without a directory specified. This option is position dependent (see Section 3.3.2, "Linking with Libraries").

-o *filename*

Use *filename* as output filename of the linker. If this option is omitted, the default filename is `a.out`.

-r
No report is made for unresolved symbols. Use this option with incremental linking.

-u *symbol*
Enter *symbol* as undefined in the symbol table. This is useful for linking from a library.

-v or **-t**
Verbose option. Print the name of each file as it is processed.

-w *n*
Give a warning level between 0 an 9 (inclusive). All warnings with a level above *n* are suppressed. The level of a message is printed in the last column of this message. If you do not use the **-w** option, the default warning level is 8.

3.3 Libraries

There are two kinds of libraries. One of them is the user library. If you make your own library of object modules, this library must be specified as an ordinary filename. The linker will not use any search path to find such a library. The file must have the extension `.a`.

Example:

```
lk88 start.obj -fobj.lnk mylib.a
```

or, if the library resides in a sub directory:

```
lk88 start.obj -fobj.lnk libs\mylib.a
```

The other kind of library is the system library. You must define system libraries with the **-l** option. With the option **-lcs** you specify the system library `libcs.a`.

3.3.1 Library Search Path

The linker searches for system library files according to the following algorithm:

1. Use the directories specified with the **-L***directory* options, in a left-to-right order. For example:

   ```
   lk88 -L..\lib -L\usr\local\lib start.obj -fobj.lnk -lcs
   ```

2. If the **-L** option is not specified without a directory, check if the environment variable C88LIB exists. If it does, use the contents as a directory specifier for library files. It is possible to specify more than one directory in the environment variable C88LIB by separating the directories with a directory separator. Valid directory separators are:

   Instead of using **-L** as in the example above, the same directory can be specified using C88LIB:

   ```
   set C88LIB=..\lib;\usr\local\lib
   lk88 start.obj -fobj.lnk -lcs
   ```

3. Search in the `lib` directory relative to the installation directory of **lk88** for library files.
   lk88.exe is installed in the directory `C:\C88\BIN`
   The directory searched for the library file is `C:\C88\LIB`

   The linker determines run-time which directory the binary is executed from to find this `lib` directory.

4. If the library is still not found, search in the processor and model specific subdirectory of the `lib` directory relative to the installation directory of **lk88** for library files. For example:

   `C:\C88\LIB\S1C88s`

The S1C88s directory is searched if the application is built in the small memory model. In general, the following directories are searched:

Directory	Application built in
S1C88s	small model
S1C88d	compact data model
S1C88c	compact code model
S1C88l	large model

For an explanation of memory models, see also Chapter 1, "C Compiler" and Section 2.7.3.5, "MODEL".

A directory name specified with the **-L***directory* option or in C88LIB may or may not be terminated with a directory separator, because **lk88** inserts this separator, if omitted.

3.3.2 Linking with Libraries

If you are linking from libraries, only those objects you need are extracted from the library. This implies that if you invoke the linker like:

```
lk88 mylib.a
```

nothing is linked and no output file will be produced, because there are no unresolved symbols when the linker searches through `mylib.a`.

It is possible to force a symbol as undefined with the option **-u**:

```
lk88 -u main mylib.a
```
(space between -u and main is optional)

In this case the symbol `main` will be searched for in the library and (if found) the object containing `main` will be extracted. If this module contains new unresolved symbols, the linker looks again in `mylib.a`. This process repeats until no new unresolved symbols are found. See also the library member search algorithm in the next section.

The position of the library is important, if you specify:

```
lk88 -lcs myobj.obj mylib.a
```

the linker starts with searching the system library `libcs.a` without unresolved symbols, thus no module will be extracted. After that, the user object and library are linked. When finished, all symbols from the C library remain unresolved. So, the correct invocation is:

```
lk88 myobj.obj mylib.a -lcs
```

All symbols which remain unresolved after linking `myobj.obj` and `mylib.a` will be searched for in the system library `libcs.a`. Note that the link order for objects, user libraries and system libraries is the order in which they appear at the command line. Objects are always linked, object modules in libraries are only linked if they are needed.

3.3.3 Library Member Search Algorithm

A library built with **ar88** always contains an index part at the beginning of the library. The linker scans this index while searching for unresolved externals. However, to keep the index as small as possible, only the defined symbols of the library members are recorded in this area.

When the linker finds a symbol that matches an unresolved external, the corresponding object file is extracted from the library and is processed. After processing the object file, the remaining library index is searched. If after a complete search unresolved externals are introduced, the library will be scanned again.

Using the **-v** option, you can follow the linker actions in respect to the libraries.

3.4 Linker Output

The linker produces an IEEE-695 object output file and, if requested, a map file, and/or a call graph file.

The linker output object is still relocatable. It is the task of the locator to determine the absolute addresses of the sections. The linker combines sections with the same name to one (bigger) output section.

The linker produces a map file if the option **-M** is specified. The name of the map file is the same as the name of the output file. The extension is `.lnl`. If no output filename is specified the default name is `a.lnl`. The map file is organized per linked object. Each object is divided in sections and symbols per section. The map file shows the relative position of each linked object from the start of the section.

The generated call graph will also be printed in the map file. The call graph contains an overview of which function calls are present. The call graph also contains information about the stack usage of the call graph. When a function is called, the stack usage before entering the function is written in front of the function name. The total stack usage of the function (including its calls) is written behind the function. The maximum stack usage of a function itself is written below the function. The number indicates the size of the stack usage (in bytes for the S1C88). See also the example.

The call graph also generates two types of messages:

- one for the detection of a recursive function call which is displayed as:

```
Call graph(s)
=============

Call graph 1:

 function
  |
  +-- function1  !! RECURSIVE !!
```

- one for the detection of a static function that is referenced / called by different call graphs.

  If a static function is called by different call graphs, the function is handled as a separate graph and is not overlayed with the different call graphs it is referenced in.

```
Call graph(s)
=============

Call graph 1:

 root1
  |
  +-- shared  !! NOT OVERLAYED !! (referenced by different call graphs)

Call graph 2:

 root2
  |
  +-- shared  !! NOT OVERLAYED !! (referenced by different call graphs)
  |
  +-- sub2
```

The command line option **-c** forces the linker to generate a separate call graph file with a compressed call graph. The filename extension of this file is `.cal`.

If the linker is used for incremental linking, the **-r** option must be used. The effect is, that unresolved symbol diagnostics will not be generated, and overlaying is not done (see Section 3.5, "Overlay Sections"). In this case, the output of the linker can be used again as input object. A call graph will always be generated.

A sample map file (`.lnl`):

```
Call graph(s)
=============

Call graph 1:

 _start ( 14 )
  |
  +-( 4 )- _exit ( 2 )
  |    |
  |    +-( 2 )
  |
  +-( 2 )- main ( 12 )
  |    |
  |    +-( 2 )- puts ( 10 )
  |    |    |
  |    |    +-( 2 )- fputc ( 8 )
  |    |    |    |
  |    |    |    +-( 2 )- _flsbuf ( 6 )
  |    |    |    |    |
  |    |    |    |    +-( 2 )- _iowrite ( 2 )
  |    |    |    |    |    |
  |    |    |    |    |    +-( 2 )
  |    |    |    |    |
  |    |    |    |    +-( 2 )- _write ( 4 )
  |    |    |    |    |    |
  |    |    |    |    |    +-( 2 )- _iowrite ( 2 )
  |    |    |    |    |    |    |
  |    |    |    |    |    |    +-( 2 )
  |    |    |    |    |    |
  |    |    |    |    |    +-( 2 )
  |    |    |    |    |
  |    |    |    |    +-( 2 )
  |    |    |    |
  |    |    |    +-( 2 )
  |    |    |
  |    |    +-( 2 )
  |    |
  |    +-( 2 )
  |
  +-( 4 )

Maximum stack usage: 14
Pool offsets
============

Pool #1: zp_ovln  (Total of  39 bytes)

                Pool: zp_ovln
                        off     siz
        puts()          0       6
        fputc()         6       7
        _flsbuf()       13      12
        _write()        25      10
        _iowrite()      35      4
Object: cstart.obj
==================
```

```
Section:abs_65534 ( Start = 0x0 )


Section:.text ( Start = 0x0 )
0x0000001c E __exit
0x00000000 E __START
Object: hello.obj
==================

Section:.text ( Start = 0x1f )
0x0000001f E _main


Section:.string ( Start = 0x0 )
Object: _puts.obj
==================

Section:.text ( Start = 0x28 )
0x00000028 E _puts
Object: _fputc.obj
===================

Section:.text ( Start = 0x78 )
0x00000078 E _fputc
Object: _iob.obj
================

Section:.near_data ( Start = 0x0 )
0x00000000 E __iob


Section:.near_bss ( Start = 0x0 )
0x00000000 E __ungetc
Object: _flsbuf.obj
====================

Section:.text ( Start = 0x0102 )
0x00000102 E __flsbuf
Object: _iowrite.obj
====================

Section:.text ( Start = 0x0314 )
0x00000314 E __iowrite
Object: _write.obj
==================

Section:.text ( Start = 0x0318 )
0x00000318 E __write
```

The addresses in the map file are offsets relative to the start of the section in the output file. For instance, section `.text` of the object module `hello.obj` starts at offset 0x1f from the output `.text` section. Function `main` also starts at offset 0x1f from the start of the resulting `.text` section. The `E` after the address indicates the label is external.

When we take the following part of the call graph,

```
+----- _write ( 4 )
       |
       +-( 2 )- _iowrite ( 2 )
       |    |
       |    +-( 2 )
       |
       +-( 2 )
```

we can see from the indentation in the structure of the tree that function `_write` calls function `_iowrite`. The total stack usage of function `_write` (including its calls) is given behind the function name:

```
 _write ( 4 )
```

To determine the total stack usage we take the maximum of the following:

1. local usage before calling a function (the first value), added to the total usage of that function (the last value):
   ```
   +-( 2 )- _iowrite ( 2 )
   ```
2. the usage of the function itself:
   ```
   |
   +-( 2 )
   ```

3.5 Overlay Sections

In order to make memory use in the static memory model more effective, the compiler generates special sections, with the overlay attribute, which must be overlayed by the linker. Each C function has its own section with local variables, temporaries etc. The linker builds a call graph to determine a valid overlay of the sections of functions which do not call each other.

For example:

```
#include <stdio.h>

void foo( int );

void
main(void)
{
        int j;
        printf( "hello\n" );
        j = 2;
        foo(j);
}
void
foo( int j )
{
        int i;
        i = j;
}
```

The linker detects that `foo` does not call `printf`, and `printf` does not call `foo`. The compiler generates an overlayable data section for the local variable `i`. `printf`, which also has local variables, gets its own overlayable data section. The linker puts the overlay sections of these two functions at the same memory area. The advantage is that the target memory is used more effectively.

3.6 Type Checking

3.6.1 Introduction

By default the compiler and the assembler generate high-level type information. Unless you disable generation of type information (**-g0**), each object contains type information of high-level types. The linker compares this type information and warns you if there are conflicts. The linker distinguishes four types of conflicts:

1. Type not completely specified (W109). Occurs if you do not specify the depth of an array, or if you do not specify arguments in one of the function prototypes. The linker does not report this type of conflicts unless you specify a warning level 9 (**-w9**), default is warning level 8.
2. Compatible types, different definitions (W110). Occurs if for instance you link a short with an int. The S1C88 takes both as 16 bits, so there will not be a problem. However, the code is not portable. Also structures or types with different names produce this warning. The warning level for this message is 8, so you can switch off this kind of message by specifying warning level 7 or less (**-w7**).
3. Signed/unsigned conflict (W111). If you link a signed int with an unsigned int, you get this message. In many cases there will be no problem, but the unsigned version can hold a bigger integer.
   The warning level of this warning is 6 and can be suppressed by specifying a warning level of 5 or less (**-w5**).
4. All other type conflicts (W112). If you get warning 112, there is probably a more serious type conflict. This can be a conflict in a function return type, a conflict in length between two built in types (short/long) or a completely different type. This warning has a level of 4, and can be switched off with warning level 3 or less (**-w3**).

3.6.2 Recursive Type Checking

The linker compares type recursively. For instance, the type of `foo`:

```
struct s1 {
     struct s2 *s2_ptr;
};

struct s2 {
     int  count;
} sample;

struct s1 foo = { &sample };
```

If you compile this source and link it with another compiled source with only `struct s2` different:

```
struct s1 {
     struct s2 *s2_ptr;
};

struct s2 {
     short count;
};

extern struct s1 foo;
```

message W112 (type conflict) will be generated. Although `struct s1` is the same in both cases, this is a real type conflict: For instance, the code "`foo.s2_ptr->count++`" produces different code in both objects.

If you have several conflicts in one symbol, the linker reports only the one with the lowest warning level. (The most serious one.)

3.6.3 Type Checking between Functions

If you use K&R style functions, it is not possible to check the type of the arguments and the number of arguments. Return types are 'int' if not specified. Prototypes are only needed if a function has a non-integer return type:

```
test2( par )
int par;
{
        test1( par );
        return test3( 1, 2 );
}
```

In this case, `test1` (defined in another source) has a return type `void`, and `test3` has a return type `int`, which is the default. At the default warning level, the linker does not report any conflict. If you should specify warning level 9 (**-w9**), the linker reports a 'not completely specified' type, because the linker is not able to check the arguments. Conflicts in return types cause real type conflicts at warning level 4.

If the source is ANSI style (which is recommended), the linker checks the types of all parameters, and the number of parameters. In this case the source of the example above looks like:

```
void test1( int );     /* ANSI style prototypes */
int  test3( int, int );

test2( int par )       /* ANSI style function definition*/
{
        test1( par );
        return test3( 1, 2 );
}
```

Another source, containing the definition of `test1` and `test3` may look like:

```
void test1( int one )
{
        /*
        **  code for function test1
        */
        .
        .
        .
}
int test3( int one, int two )
{
        /*
        **  Code for function test3
        */
        .
        .
        .
}
```

Prototypes are only needed for functions which are referenced before they are defined within one source. However, it is a good practice to include a prototype file with prototypes of all the functions in a file. If you do so, type checking for functions is done by the compiler. Nevertheless, if you do not compile all sources after you have changed the prototype file, the linker will report the type conflict.

It is possible to add ANSI style prototypes to K&R style C-code. In this case full type checking for functions becomes available. To accomplish this, make a new header file with all prototypes for all functions in your application. Include this file in each source, or tell the compiler to include it for you by means of the option **-H**:

```
cc88 -c -Hproto.h *.c
```

3.6.4 Missing Types

In C you are allowed to define pointers to unspecified objects. The linker is not able to check such types. For instance:

```
struct s1 {
        struct s2 *s2_ptr;
};

struct s1 foo;
```

The structure `s2` is not specified. Because the linker is not able to check whether `struct s2` is the same in all sources, a warning at level 9 will be generated:

```
lk88 W102 (9) <name>: Incomplete type specification, type index = T101
```

It is possible that the `struct s2` is known in an other source. If this source uses variable `foo`, a second message is generated, reporting a level 9 type conflict:

```
lk88 W109 (9) <f1>: Type not completely specified for symbol <foo> in <f2>
```

Because the type definition is not complete, the first warning reports that the linker cannot check the type, although this is allowed in C. This message is given once for each object for each incomplete type. The second warning reports a difference in types, an incomplete type versus a complete type. Note that al these warnings are only generated if you specify warning level 9 (**-w9**).

3.7 Linker Messages

There are four kinds of messages: fatal messages, error messages, warning messages and verbose messages. Fatal messages are generated if the linker is not able to perform its task due to the severity of the error. In those situations, the exit code will be 2. Error messages will be reported if an error occurred which is not fatal for the linker. However, the output of the linker is not usable. The exit code in case of one or more error messages will be 1. Warning messages are generated if the linker detects potential errors, but the linker is unable to judge those errors. The exit code will be 0 in this case, indicating a usable `.out` file. Of course, if the linker reports no messages at all, the exit code is 0 also.

Each linker message has a built-in warning level. With option **-w***x* it is possible to suppress messages with a warning level above *x*.

Verbose messages are generated only if the verbose option (**-v**) is on. They report the progress of the link process.

Linker messages have the following layout:

```
S1C88 object linker vx.y rz        SN000000-000 (C)year Tasking Software BV
lk88 W112 a.obj: Type conflict for symbol <f> in b.obj                (4)
```

The first line shows the banner of the S1C88 linker. The second line reports a type conflict in the file `a.obj`. Apparently there is a conflicting type definition of the function `f` in module `b.obj`. The number at the end of the line '(4)', shows the warning level.

There are four message groups:

1. Fatal (always level 0):
   - Write error
   - Out of memory
   - Illegal input object
2. Error (always level 0):
   - Unresolved symbols (and no incremental linking)
   - Can't open input file
   - Illegal recursive use of an non reentrant function
3. Warning (levels from 1 to 9):
   - Type conflict between two symbols
   - Illegal option (Ignored)
   - No system library search path, and system library requested
4. Verbose (level not relevant, only given with option **-v**):
   - Extracting files from a library
   - Current file/library name
   - pass one or pass two
   - Rescanning library for new unresolved symbols
   - Cleaning up temp files
   - warning level

CHAPTER 4 LOCATOR

4.1 Overview

This chapter describes the S1C88 locator.

The task of the locator is to locate a `.out` file, made by **lk88**, to absolute addresses. In an embedded environment an accurate description of available memory and information about controlling the behavior of the locator is crucial for a successful application. For example, it may be necessary to port applications to processors with different memory configurations, or it may be necessary to tune the location of sections to take full advantage of fast memory chips. To perform its task the locator needs a description of the derivative of the S1C88 used. The locator uses a special language for this description: DELFEE, which stands for DEscriptive Language For Embedded Environments. This steering language is used in a special file, which is called the description file. See Chapter 5, "DEscriptive Language For Embedded Environments", for detailed information.

The description file is an optional parameter in the locator invocation. Without a description file name on the command line, or without the **-d** option, the locator searches the file `s1c88.dsc` in the current directory or in directory `etc` in the S1C88 product tree.

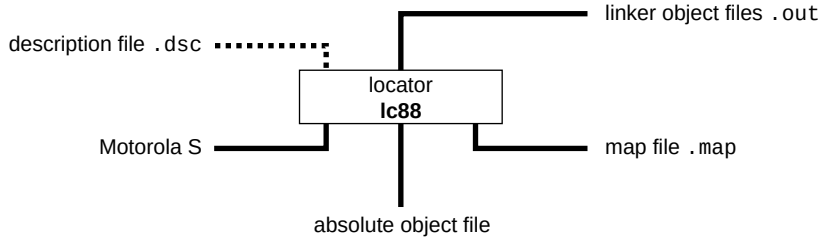


Fig. 4.1.1 Locator

4.2 Invocation

The invocation of the locator is:

lc88 [*option*]... [*file*]...

Options may appear in any order. Options start with a '**-**'. They may be combined: **-eM** is equal to **-e -M**. Options that require a filename or a string may be separated by a space or not: **-o***name* is equal to **-o** *name*.

file may be any file with a `.out` or `.dsc` extension.

The locator recognizes the following options:

Options Summary

Option	Description
-M	Produce a locate map file (`.map`)
-S *space*	Generate specific *space*
-V	Display version header only
-d *file*	Read description file information from *file*, '-' means `stdin`
-e	Clean up if erroneous result
-err	Redirect error messages (`.elc`)
-f *file*	Read command line information from *file*, '-' means `stdin`
-f *format*	Specify output format
-o *filename*	Specify name of output file
-p	Make a proposal for a software part on `stdout`
-v	Verbose option. Print name of each file as it is processed
-w *n*	Suppress messages above warning level *n*.

4.2.1 Detailed Description of Locator Options

-M

Produce a locate map (`.map`).

-S *space*

With this option you can generate a specific output file for a specified *space* instead of generating an output file containing all spaces. *space* is the name of a space from a `.dsc` file.

-V

With this option you can display the version header of the locator. This option must be the only argument of **lc88**. Other options are ignored. The locator exits after displaying the version header.

-d *file*

Read description file information from *file* instead of a `.dsc` file. If *file* is a '**-**', the information is read from standard input.

-e

Remove all locate products such as temporary files, the resulting output file and the map file, in case an error occurred.

-err

Redirect error messages to an error file with the extension `.elc`.

-f *file*

Read command line information from *file*. If *file* is a '-', the information is read from standard input. You need to provide the EOF code to close `stdin`. *file* may not be a number in the range 0–3, because these numbers are used to specify an output format.
Use *file* for command line processing. To get around the limits on the size of the command line, it is possible to use command files. These command files contain the options that could not be part of the real command line. Command files can also be generated on the fly, for example by the make utility. More than one **-f** option is allowed.
Some simple rules apply to the format of the command file:

1. It is possible to have multiple arguments on the same line in the command file.
2. To include whitespace in the argument, surround the argument with either single or double quotes.
3. If single or double quotes are to be used inside a quoted argument, we have to go by the following rules:
   a. If the embedded quotes are only single or double quotes, use the opposite quote around the argument. Thus, if a argument should contain a double quote, surround the argument with single quotes.
   b. If both types of quotes are used, we have to split the argument in such a way that each embedded quote is surrounded by the opposite type of quote.

   Example:
   ```
   "This has a single quote ' embedded"
   ```
   or
   ```
   'This has a double quote " embedded'
   ```
   or
   ```
   'This has a double quote " and a single quote '"' embedded"
   ```
4. Some operating systems impose limits on the length of lines within a text file. To circumvent this limitation it is possible to use continuation lines. These lines end with a backslash and newline. In a quoted argument, continuation lines will be appended without stripping any whitespace on the next line. For non-quoted arguments, all whitespace on the next line will be stripped.

Example:

```
"This is a continuation \
line"
        → "This is a continuation line"

control(file1(mode,type),\
file2(type))
        → control(file1(mode,type),file2(type))
```

5. It is possible to nest command line files up to 25 levels.

-f *format*

Specify output format. *format* can be one of the following output formats:

1 = IEEE Std. 695 (Default)
2 = Motorola S records

The default output format is IEEE Std. 695 (**-f1**), which can directly be used by the debugger. The other output formats can be used for loading into a PROM-programmer.

-o *filename*

Use *filename* as output filename of the locator. If this option is omitted, the default filename depends on the output format specified:

Format	Default output name
1	`a.abs`
2	`a.sre`

-p

Make a proposal for a software part in a description file on standard output.

-v

Verbose option. Print the name of each file as it is processed.

-w *n*

Give a warning level between 0 an 9 (inclusive). All warnings with a level above *n* are suppressed. The level of a message is printed in the last column of this message. If omitted, the warning level defaults to 8.

4.3 Getting Started

The locator invocation is normally done via the control program. This control program hides the locator phase completely. In this section you will invoke the locator as a separate tool in order to get a better understanding of the use of options and the description file.

You can find a more detailed description of the descriptive language for embedded environments (DELFEE) in Chapter 5, "DEscriptive Language For Embedded Environments".

If you want to locate the `calc` demo, you need the relocatable demo file `calc.out` as input for the locator. You can generate this file by copying the contents of the directory `examples\asm` to your working directory, and invoke the control program:

```
cc88 -cl -M -Ms -nolib startup.asm _copytbl.asm watchdog.asm
     calc.asm -o calc.out
```

Be sure that the `bin` directory of the S1C88 tools is in the search path. The option **-cl** tells the control program to stop after linking and to suppress the locating phase. The file you made by this command is the complete demo, but still in a relocatable form. Now, you can locate this relocatable file `calc.out` to absolute addresses by typing:

```
lc88 -M calc.out -ds1c88316.dsc
```

The **-M** option causes **lc88** to make a map file. The default output file format is IEEE-695 (**-f1** option). Since you did not specify an output name, the default output name `a.abs` will be generated. (For **-f1** the default is `a.abs` and for **-f2** the default is `a.sre`) After the invocation, the locator has generated two files:

- `a.abs`, The IEEE 695 output file
- `a.map`, The locate map file

If you want to give the output file a specific name, you must use the **-o***file* option:

```
lc88 -M calc.out -o calc.abs -ds1c88316.dsc
```

You may need to adjust the description file. In a description file you can change the locating algorithm of the locator. If you do not specify a description file (argument of **-d** option), the locator uses the file `s1c88.dsc` from the `etc` sub directory (in the S1C88 product tree). With the **-d** option given above you specify the `s1c88316.dsc` description file. If you do not want to change this original description file (which is advisable), make a copy of file `s1c88316.dsc` to your working directory.

You can change the copy of the description file. Everything after a comment (`//`) until the end of the line is ignored. As an example, change the lines:

```
amode code {
        section selection=x;
        section selection=r;
        copy;
        table;
}
```

into:

```
amode code {
        section .text;
        section .ptext;
        copy;
        table;
}
```

The effect will be that the location order of the sections `.text` and `.ptext` is now forced to be fixed.

Locate again to see the effect. The modified description file `s1c88316.dsc` in your working directory will be found before the original version in the `etc` directory. Because you want to compare the map files, choose another output name:

```
lc88 -M calc.out -ocalc_o.abs -ds1c88316.dsc
```

Now you can compare `calc.map` and `calc_o.map`.

If you want to choose between a description file with and without the changes you made, you must rename the `s1c88316.dsc` in your working directory to, for example, `order.dsc`. If you want the changed version of the description, you can invoke the locator as follows:

```
lc88 -M -d order calc.out -ocalc_o.abs
```

The space between **-d** and **order** is optional. If you do install `order.dsc` in the `etc` subdirectory, you can use the option **-dorder** from any working directory.

If you want to know more about the locate language DELFEE, read Chapter 5.

4.4 Calling the Locator via the Control Program

It is recommended to call the locator via the control program **cc88**. The control program translates certain options for the locator (e.g., **-srec** to **-f2**). Other options (such as **-M**) are passed directly to the locator. Typical, you can use the control program to get an `.abs` file directly from `.c`, `.src`, `.asm` or `.obj` files. The invocation:

```
cc88 -M -Ms -g -nolib startup.asm _copytbl.asm watchdog.asm calc.asm
     -o calc.abs s1c88316.dsc
```

builds an absolute demo file called `calc.abs` ready for running via the debugger.

4.5 Locator Output

The locator produces an absolute file and, if requested, a map file and/or an error file. The output file is absolute and in Motorola S-record format or in IEEE-695 format, depending on the usage of the **-f** option. The default output name is `a.sre` or `a.abs`, respectively.

The map file (**-M** option) always has the same basename as the output object file, with an extension `.map`. The map file shows the absolute position of each section. External symbols are listed with their absolute address, both sorted on address and sorted on symbol.

The error output file (**-err** option) has the same name as the object output file, but with extension `.elc`. Errors occurred before the **-err** option is evaluated are printed on stderr.

4.6 Locator Messages

There are four kinds of messages: fatal messages, error messages, warning messages and verbose messages. Fatal messages are generated if the locator is not able to continue with its task due to the severity of the error. In those situations, the exit code will be 2. Error messages will be reported if an error occurred, not fatal for the locator. However, the output of the locator is not usable. The exit code in case of one or more error messages will be 1. Warning messages are generated if the locator detects potential errors, but the locator is unable to judge those errors. The exit code will be 0 in this case, indicating a usable `.abs` file. Of course, if the locator reports no messages, the exit code is also 0.

Each locator message has a built-in warning level. With option **-w***x* it is possible to suppress messages with a warning level above *x*.

Verbose messages are generated only if the verbose option (**-v**) is on. They report the progress of the locate process.

Locator messages have the following layout:

```
S1C88 locator vx.y rz             SN000000-127 (C)year Tasking Software BV
lc88 W112 (3) calc.out: Copy table not referenced, initial data is not copied
```

The first line shows the locator banner. (Suppressed if the locator invocation is done by the control program.) The second line shows the warning. The number after the warning number shows the warning level.

4.7 Address Space

Figures 4.7.1 and 4.7.2 show the different address space mappings of the S1C88.

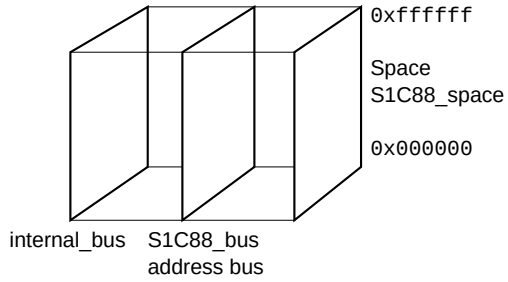


Fig. 4.7.1 S1C88 physical address space mapping

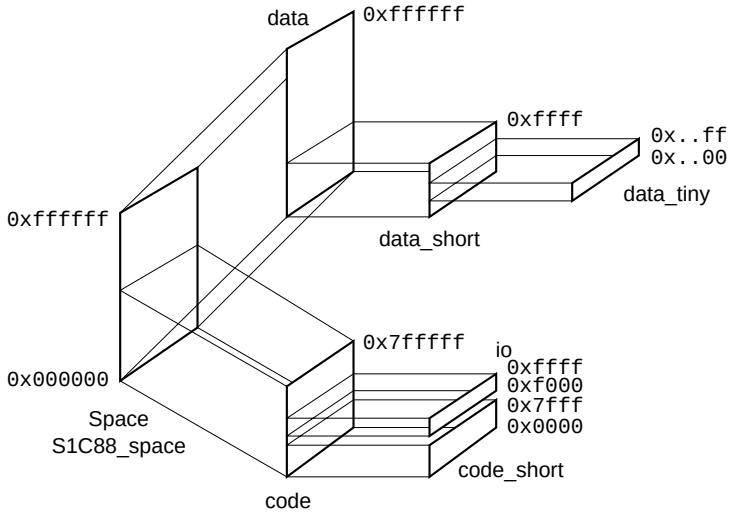


Fig. 4.7.2 S1C88 virtual address space mapping

4.8 Copy Table

One of the actions with the process initialization is copy data from ROM to RAM, and initialize memory with the CLEAR attribute. The locator generates a copy table for each process. The copy table can be referenced by label `__lc_cp`. One entry in the copy table has the following layout (see `locate.h`, delivered with the C compiler):

```
typedef struct cp_entry {
        char                    cp_actions;     /* 1 byte */
        _huge unsigned char     *cp_destin;     /* 3 byte address */
        _huge unsigned char     *cp_source;     /* 3 byte address */
        unsigned long           cp_length;      /* 4 byte length */
} cp_entry_t;
```

The first member, `cp_actions`, defines what action you must perform with the current entry. Actions are organized as a bit per action:

value 0	Reached end of the table.
CP_COPY (value 1)	Copy from `cp_source` to `cp_destin` over `cp_length` bytes.
CP_BSS (value 2)	Clear memory from `cp_destin` over `cp_length` bytes.

Table entries are generated as follows:

- one entry for each section with the CLEAR attribute
- one entry for each section with the INIT attribute
- one 'zero' entry to indicate the end-of-table

If there is nothing to do (no sections to clear and no data to copy) the copy table has only one action entry with value zero.

At C level, the copy table can be declared as:

```
cpt_t   _lc_cp;
```

And accessing a member of entry `x` becomes:

```
_lc_cp[ x ].cp_actions;
```

If label `__lc_cp` is not used, the table is not generated.

4.9 Locator Labels

The locator assigns addresses to the following labels when they are referenced:

`__lc_cp` : Start of copy table The copy table gives the source and destination addresses of sections to be copied. This table will be generated by the locator only if this label is used.

`__lc_bs` : Begin of stack space (using keyword `stack`).

`__lc_es` : End of stack space. Initialization of stack pointer.

`__lc_b_`*name* : Begin of section *name*.

`__lc_e_`*name* : End of section *name*.

`__lc_u_`*name* : User defined label. The label must be defined in the description file. For example:

```
label mylab;
```

`__lc_ub_`*name* : Begin of user defined label. The label must be defined in the description file. For example:

```
label mybuffer length=100;
```

`__lc_ue_`*name* : End of user defined label.

4.9.1 Locator Labels Reference

This section contains a description of all locator labels. Locator labels are labels starting with **__lc_**. They are ignored by the linker and resolved at locate time. Some of these labels are real labels at the beginning or the end of a section. Other labels have a second function, these labels are used to address locator generated data. The data is only generated if the label is used.

Because labels that start with **__lc_** are treated differently in both the linker and the locator, you can only use this type of labels as references, not as definitions.

Note: At C level, all locator labels start with one leading underscore (the compiler adds another underscore '_').

__lc_b_*section*, __lc_e_*section*

Syntax:

```
extern unsigned char   __lc_b_section[ ];
extern unsigned char   __lc_e_section[ ];
```

Description:

You can use the general locator labels **__lc_b_***section* and **__lc_e_***section* to obtain the addresses of section *section* in a program. The **b** version points to the start of the section, while the **e** version points to its end.

You can replace the dot before a section name by an underscore (_), making it possible to access these labels from 'C'. This convention introduces a possible name conflict. If, for example, both sections `.text` and `_text` exist, the general label `__lc_b__text` is set to the start of `_text`. The label for section `.text` is only usable at assembly level with its real name. Of course, you should avoid such a conflict by not using section names with a leading underscore.

Example:

```
printf( "Text size is 0x%x\n", _lc_e__text - _lc_b__text );
```

__lc_bh, __lc_eh

Syntax:

```
extern unsigned char   __lc_bh[ ];
extern unsigned char   __lc_eh[ ];
```

Description:

All locator **h** labels are related to the heap. You can allocate a heap by defining it in a cluster description. See also the Delfee keyword **heap**.

__lc_bh is a label at the begin of the heap. At 'C' level **_lc_bh** represent the heap. The label is defined as a char array, but an array of any basic type will do. **__lc_eh** represents the end of the heap.

Example:

Heap definition:

```
block total_range {
        .
        .
        cluster ram {
          amode data {
                heap length = 200;
                .
          }
        }
        .
}
```

sbrk code:

```
extern unsigned char _lc_bh[ ];
extern unsigned char _lc_eh[ ];

static char *
sbrk( long length ) {
        .
        .

if ( (lastmem + length) > _lc_eh ) {
        return (char *) -1; /* overflow */
}
```

__lc_bs, __lc_es

Syntax:

```
extern unsigned char   _lc_bs[ ];
extern unsigned char   _lc_es[ ];
```

Description:

All locator **s** labels are related to the stack. You can allocate a stack by defining it in a cluster description. See also the Delfee keyword **stack**.

__lc_bs is a label at the begin of the stack. At 'C' level **_lc_bs** represent the stack. The label is defined as a char array, but an array of any basic type will do. **__lc_es** represents the end of the stack. Because **__lc_es** is on a higher address than **__lc_bs** and because the stack for the S1C88 grows to lower addresses, the stack actually starts at the label **__lc_es** and ends at **__lc_bs**.

Example:

Stack definition:

```
block total_range {
        cluster ram {
          amode data {
                stack length = 100;
                .
          }
        }
}
```

Stack initialization:

```
__START:
        LD SP,#__lc_es  ; set stack pointer to
                        ; begin of stack space
```

__lc_cp

Syntax:

```
extern char *_lc_cp;
```

Description:

The copy table is generated per process. Each entry in this table represents a copy or clearing action. Entries for the table are automatically generated by the locator for:

- All sections with attribute b, which must be cleared at startup time: a clearing action.
- All sections with attribute i, which must be copied from rom to ram at program startup: a copy action.

The layout of the copy table is described in Section 4.8, "Copy Table". Type `cpt_t` is defined in `locate.h`.

__lc_u_*identifier*

Syntax:

```
extern int  _lc_u_identifier[ ];
```

Description:

This locator label can be defined by the user by means of the Delfee keyword **label**. This label must be defined in the Delfee file without the prefix **__lc_u_**. From assembly the label can be referenced with the prefix **__lc_u_**, from C with the prefix **_lc_u_** (one leading underscore).

Example:

In description file:

```
block total_range {
        cluster ram {
          amode data {
                label bstart;
                section text;
                label bend;
          }
        }
        .
        .
        .
}
```

From C:

```
#include <stdio.h>
extern int _lc_u_bstart[];
extern int _lc_u_bend[];
int main()
{
  printf( "Size of cluster ram is %d\n",
               (long)_lc_u_bend -
               (long)_lc_u_bstart );
}
```

__lc_ub_*identifier*, __lc_ue_*identifier*

Syntax:

```
extern int  _lc_ub_identifier[ ];
extern int  _lc_ue_identifier[ ];
```

Description:

These locator labels can be defined by the user by means of the Delfee keywords **reserved label=**. The locator labels specify the begin and end of a reserved area. The *identifier* is the name for the reserved area and must be defined in the Delfee file without the prefix **__lc_ub_** or **__lc_ue_**. From assembly the labels can be referenced with the prefix **__lc_ub_** and **__lc_ue_**, from C with the prefix **_lc_ub_** and **_lc_ue_** (one leading underscore).

Example:

In description file:

```
block total_range {
        cluster ram {
                attribute w;
                amode data {
                        section selection=w;
                        reserved label=xvwbuffer length=0x10;
                        // Start address of reserved area is
                        // label __lc_ub_xvwbuffer
                        // End address of reserved area is
                        // label __lc_ue_xvwbuffer
                }
        }
}
```

From C:

```
#include <stdio.h>
extern int _lc_ub_xvwbuffer[];
extern int _lc_ue_xvwbuffer[];
int main()
{
  printf( "Size of reserved area xvwbuffer is %d\n",
               (long)_lc_ue_xvwbuffer -
               (long)_lc_ub_xvwbuffer );
}
```

CHAPTER 5 DESCRIPTIVE LANGUAGE FOR EMBEDDED ENVIRONMENTS

5.1 Introduction

In an embedded environment an accurate description of available memory and control over the behavior of the locator is crucial for a successful application. For example, it may be necessary to port applications to processors with different memory configurations, or it may be necessary to tune the location of sections to take full advantage of fast memory chips.

For this purpose the DELFEE language, which stands for DEscriptive Language For Embedded Environments, was designed.

5.2 Getting Started

5.2.1 Introduction

This section gives a general introduction about the DELFEE description language. The goal is to give you an overview and some basic knowledge what the DELFEE description language is about, and how a basic description file looks. A more detailed description and examples are given in the following sections.

5.2.2 Basic Structure

The DELFEE language describes where code or data sections should be placed on the actual memory chips. This language has to define the interface between a virtual world (the software) and a physical world (the hardware configuration).

On the one side, in the virtual world, there are the code and data sections which are described by the assembly language. Sections can have names, attributes like writable or read-only and can have an address in the addressing space or an addressing mode describing the range of the address space in which they may be located.

On the other side, the physical world, the actual processor is present which reads instructions from memory chips and interprets these instructions. With the DELFEE language you can instruct the locator to place the code and data sections at the correct addresses, taking into account things like the type of memory chip (rom/ram, fast/slow), availability of memory, etc. The DELFEE language gives the possibility to tune the same application for different hardware configurations.

In the DELFEE language the interface between virtual and physical world is described in three parts:

1. software part (`*.dsc`)
   The software part belongs to the virtual world and describes the ordering of the data and code sections. The software part may vary for different applications and can even be empty.
2. cpu part (`*.cpu`)
   The cpu part is the interface between the virtual world and the real world. It contains the application independent part of the virtual world (the address translation of addressing modes to the addressing space), and the configuration independent part of the physical world (on-chip memory, address busses). The cpu part is independent of application and configuration.
3. memory part (`*.mem`)
   The memory belongs to the physical world. It contains the description of the external memory. The memory part may vary for different configurations and can even be empty (if there is no external memory).

Notice that the software part and the memory part can be empty, but that the cpu part must always be defined.

The DELFEE language is used in a special file, which is called the description file. In the DELFEE description language the different parts are defined with the following syntax:

```
software {
        layout {
                // ordering of sections
        }
}

cpu {
        // mapping of addressing modes to address space
        // defining address space
        // mapping of address space to actual busses
        // defining on-chip memory
}

memory {
        // description of external memory
}
```

For convenience the cpu part and the memory part can be placed in different files, which makes it possible to have different layout parts for different applications and different memory parts for different configurations. The files can be included using the syntax:

```
cpu filename   // include cpu part defined in file filename
mem filename   // include memory part defined in file filename
```

5.3 CPU Part

5.3.1 Introduction

The cpu part contains the application and configuration independent part of the description file. This part defines the translations of the addresses from the assembler language (virtual addresses) all the way down to the chips (physical addresses). To describe the translations, DELFEE recognizes four main levels:

1. addressing mode(s) definitions. Addressing modes are subsets of an address space. They define address ranges within an address space.
2. address space(s) definitions. The address space is the total range of addresses available.
3. bus(ses) definitions.
4. (on-chip) memory chips definitions.

The address translation is defined from addressing mode via space and bus to the chip. The addressing modes and the busses can be nested, the space and the chip cannot.

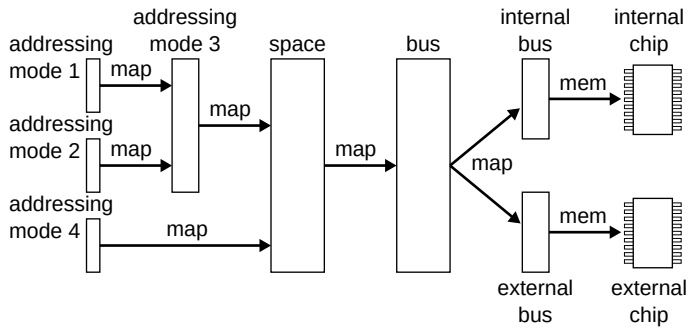


Fig. 5.3.1.1 Address translation

The addressing modes and addressing spaces belong to the virtual part, the busses and chips belong to the physical part. The following sections describe the address space and the addressing modes which are subsets of the address space. Then a description of the physical side (hardware configuration) follows, describing the busses and chips that are available.

The following example illustrates how a cpu part could look like. It is a fictitious example, mainly used to illustrate the definitions. You should be able to recognize the addressing mode definitions, address space definition, bus definitions and on-chip memory definition. Each definition is explained in the following sub-sections.

```
cpu {
        //
        // addressing mode definitions
        //
        amode near_code {
                attribute Y1;
                mau 8;
                map src=0 size=1k dst=0 amode = far_code;
        }
        amode far_code {
                attribute Y2;
                mau 8;
                map src=0 size=32k dst=0 space = address_space;
        }
        amode near_data {
                attribute Y3;
                mau 8;
                map src=0 size=1k dst=0 amode = far_data;
        }
        amode far_data {
                attribute Y4;
                mau 8;
                map src=0 size=32k dst=32k space = address_space;
        }
        //
        // space definitions
        //
        space address_space {
                mau 8;
                map src=0   size=32k dst=0   bus = address_bus label = rom;
                map src=32k size=32k dst=32k bus = address_bus label = ram;
        }
        //
        // bus definitions
        //
        bus address_bus {
                mau 8;
                mem addr=0   chips=rom_chip;
                map src=0x100 size=0x7f00  dst=0x100 bus = external_rom_bus;
                mem addr=32k chips=ram_chip;
                map src=0x8100 size=0x7f00 dst=0x100 bus = external_ram_bus;
        }
        //
        // internal memory definitions
        //
        chips rom_chip  attr=r mau=8 size=0x100;  // internal rom
        chips ram_chip  attr=w mau=8 size=0x100;  // internal ram
}
```

5.3.2 Address Translation: map and mem

In DELFEE there are two ways to describe a memory translation between two levels (the source level and the destination level):

1. **map** keyword. This is for address translations between amodes, spaces, busses (not chips).
2. **mem** keyword. This describes the address translation between bus and chip. **mem** is a simplified case of **map**.

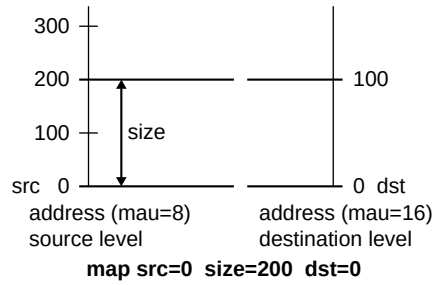


Fig. 5.3.2.1 Map address translation

The generalized syntax for the map definition is (see Figure 5.3.2.1):

map **src=***number* **size=***number* **dst=***number* *destination_type=destination_name optional_specifiers***;**

where,

src start address of the source level. In case of an address translation between amodes and spaces, the source level is the amode and the destination level is the space.

size length of the source level.

dst start address at the destination level.

destination_type

the destination type depends on the context the mapping is used in and can have three different types:

1. **amode** allowed in context: amode.
2. **space** allowed in context: amode.
3. **bus** allowed in context: space, bus.

optional_specifiers

The optional identifiers are also dependent of the context they are used in:

1. **label** Only allowed in space context and needed as a reference for the block definition in the software part (see Section 5.4.5).

   label = *name* **;**

2. **align** This indicates that every section will be aligned at the specified value.

   align = *number* **;**

3. **page** This indicates that every section should be within a given page size.

   page = *number* **;**

Both the source level and the destination level have an address range that is expressed in a number of Minimum Addressable Units (MAU, the minimal amount of storage, in bits, that is accessed using an address). The mapping only describes the range and the destination of the address mapping, the actual transformation also depends on the memory unit that an address can access. If a source level with a minimum addressable unit of 8 bits (`mau=8`) maps to a destination level with a minimum addressable unit of 16 bits (`mau=16`), the size of the destination level, expressed in address range, is half the original size. So, according to Figure 5.3.2.1, the size of the destination level is 100.

If a map is present from *level1* down to *level2*, the map definition works as follows:

*end_address of level2 = dst + (size * mau of level1 / mau of level2)*

The **mem** description is actually a simplified case of the **map** description. The length of the address translation is taken from the chip size, the destination address is always zero. It is used to map a bus to a chip.

The syntax is:

mem addr=*number* **chips=***name***;**

where,

addr start address location of a chip.

chips the name of the chip that is located at address number.

5.3.3 Address Spaces

The link between the virtual and the physical world is the description of the address space and the way it maps onto the internal address busses.

The address space is defined by the complete range of addresses that the instruction set can access. Some instruction sets support multiple address spaces (for example a data space and a code space).

An address space is described by the syntax:

```
space name {
        mau number;
        map src=number size=number dst=number bus=bus_name label=name;
            //   :
            // more maps
}
```

where,

space defines the name by which the space can be referenced in the description file.

mau the Minimum Addressable Unit, meaning the minimum amount of storage (in bits) that is accessed using an address.

map this specifies the mapping of a range of addresses in the address space to a bus defined by *bus_name*. The range of addresses is defined by **src** and **length**, the offset on the bus is defined by **dst**. (The bus you map the address space on, may have a different MAU, which will lead to another length of the range of the bus). An address space can only map onto a bus.

Usually an address in the address space corresponds to the same address on the bus. In that case **src** and **dst** have the same value.

In the previous example there is one space definition:

```
space address_space {
        mau 8;
        map src=0   size=32k dst=0   bus=address_bus label=rom;
        map src=32k size=32k dst=32k bus=address_bus label=ram;
}
```

In this example the space is named `address_space`. Note that the **amod** definitions use this name as destination for their mappings. The minimum addressable unit (MAU) is set to 8 bits. The labels `rom` and `ram` are used by **block** definitions in the software part which are discussed in Section 5.4.5.

5.3.4 Addressing Modes

Addressing modes define address ranges in the addressing space. Addressing modes usually have a special characteristic, like bitaddressable part of memory, parts especially for code sections, zero pages, etc. The addressing modes are defined by the instruction set. The syntax of defining an addressing mode in the DELFEE language is:

An address space is described by the syntax:

amode *name* **{**
    mau *number***;**
    attr **Y***number***;**
    map **src=***number* **size=***number* **dst=***number* **amode**|**space=***name***;**
}

where,

amode the name by which the addressing mode can be referenced. In the object file the addressing mode of a section is encoded with an **Y***number*. This means that the *name* given to the addressing mode has only meaning within the description file, not to the sections!

mau the Minimum Addressable Unit, meaning the minimum amount of storage (in bits) that is accessed using an address.

attr Y the addressing mode number. Code or data sections (generated by the assembler) all have a number specifying the addressing mode they belong to. In the DELFEE description file this number is used to identify the addressing mode. This number must never be changed, because the interpretation of the sections will get mixed up.

map defines the mapping of the addressing mode to another addressing mode (**amode**) or an address space (**space**).

Below is an example of two addressing mode definitions:

```
amode near_data {
        attribute Y3;
        mau 8;
        map src=0 size=1k dst=0 amode=far_data;
}
amode far_data {
        attribute Y4;
        mau 8;
        map src=0 size=32k dst=32k space=address_space;
}
```

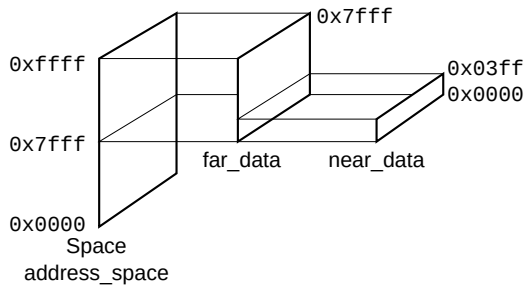


Fig. 5.3.4.1 Addressing mode mapping

In this example the addressing modes are named `near_data` and `far_data`. They are identified by the addressing mode numbers `Y3` and `Y4` respectively. The minimum addressable unit (MAU) is set to 8 bits. Addressing mode `near_data` maps on addressing mode `far_data`, and `far_data`, in its turn, maps on address space `address_space`. `address_space` is the space as discussed in the previous section.

5.3.5 Busses

The **bus** keyword describes the bus configuration of a cpu. In essence it describes the address translation from the address space to the chip. The syntax is:

bus *name* **{**
mau *number*;
map **src**=*number* **size**=*number* **dst**=*number* **bus**=*name*;
mem **addr**=*number* **chips**=*name*;
}

where,

bus the name by which the bus can be referenced.

mau the Minimum Addressable Unit, meaning the minimum amount of storage (in bits) that is accessed using an address.

map mapping to another bus.

mem mapping to a memory chip.

Below is an example of a bus definition:

```
bus address_bus {
        mau 8;
        mem addr=0   chips=rom_chip;
        map src=0x100 size=0x7f00  dst=0x100 bus=external_rom_bus;
        mem addr=32k chips=ram_chip;
        map src=0x8100 size=0x7f00 dst=0x100 bus=external_ram_bus;
}
```

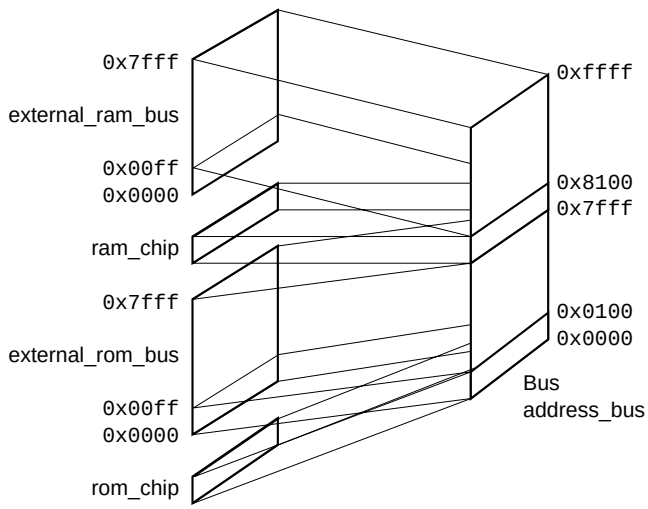


Fig. 5.3.5.1 Bus mapping

In this example the address bus is named `address_bus`. The minimum addressable unit (MAU) is set to 8 bits. The internal memory chip `rom_chip` is located at address `0` of the bus, and the chip `ram_chip` is located at address `32k`.

Two address mappings to other busses are present: one to `external_rom_bus` and one to `external_ram_bus`.

The first mapping translates addresses `0x100-0x7ff` of `address_bus` (`src=0x100 size=0x7f00`) onto addresses of `external_rom_bus` starting at address `0x100` (`dst=0x100`).

The second mapping translates addresses `0x8100-0xffff` of `address_bus` (`src=0x8100 size=0x7f00`) onto addresses of `external_ram_bus` starting at address `0x100` (`dst=0x100`). Note that the second mapping maps to RAM, not ROM. That is why both destination addresses are the same.

5.3.6 Chips

The **chips** keyword describes the memory chip. The syntax is:

chips *name* **attr=***letter_code* **mau=***number* **size=***number***;**

where,

chips the name by which the chip can be referenced.

attr defines the attributes of the chip with a letter code.

letter_code one of the following attributes:
- **r** read-only memory.
- **w** writable memory.
- **s** special memory (it must not be located).

mau the Minimum Addressable Unit, meaning the minimum amount of storage (in bits) that is accessed using an address.

size the size of the chip (address range from 0–*size*).

Below is an example of two chip definitions:

```
chips rom_chip  attr=r mau=8 size=0x100;  // internal rom
chips ram_chip  attr=w mau=8 size=0x100;  // internal ram
```

In this example the chips are named `rom_chip` and `ram_chip`. The minimum addressable unit (MAU) is set to 8 bits. The size of both chips is `0x100` MAUs (= 256 bytes). Chip `rom_chip` is read-only and chip `ram_chip` writable, as you would expect with ROM and RAM.

5.3.7 External Memory

With the syntax described in the previous sections it would be possible to define mappings from an address space to external memory chips (DELFEE does not actually know, or care, if memory is on-chip). However, this is not advisory. For maintenance and flexibility reasons it is better to keep the internal (static) memory part apart from the external (variable) memory part. Section 5.5, "Memory Part", describes how to deal with external memory.

In the cpu part you only have to define a mapping to an external bus, which can later be defined in the memory part. The following example contains references to two external busses: `external_ram_bus` and `external_rom_bus`.

```
bus address_bus {
        mau 8;
        mem addr=0   chips=rom_chip;
        map src=0x100 size=0x7f00  dst=0x100 bus=external_rom_bus;
        mem addr=32k chips=ram_chip;
        map src=0x8100 size=0x7f00 dst=0x100 bus=external_ram_bus;
}
```

5.4 Software Part

5.4.1 Introduction

The software part has two main parts:

1. load_mod
2. layout description

```
software {
        load_mod start = start_label;

        layout {
                // ordering of sections
        }
}
```

5.4.2 Load Module

The keyword **load_mod** defines the program start label. The program start label is the start of the code and the reset vector should point to this label. The locator generates a warning if this label is not referenced.

```
load_mod start = start_label;
```

5.4.3 Layout Description

First of all, the layout definition can be omitted. If you omit the layout definition, the locator will generate a layout definition based on the DELFEE description of the amodes (addressing modes) in the cpu part (See Section 5.3). However this does not allow you to control the order in which sections (like stack and heap) are located. If you define the layout part, the locator uses this description.

The layout part is probably the most difficult part of the DELFEE language. It is designed to give the locate algorithm the information it needs to locate the sections correctly. Through some examples you will be shown how to influence the locate algorithm using the DELFEE language.

To give you an idea of where all this will lead to, an example of a layout part is given:

```
layout {
        space address_space {
                block rom {
                        cluster first_code_clstr {
                                attribute i;
                                amode near_code;
                                amode far_code;
                        }
                        cluster code_clstr {
                                attribute r;
                                amode near_code {
                                        section selection=x;
                                        section selection=r;
                                }
                                amode far_code {
                                        table;
                                        section selection=x;
                                        section selection=r;
                                        copy;      // locate rom copies here
                                }
                        }
                }
```

```
                block ram {
                        cluster data_clstr {
                                attribute w;
                                amode near_data {
                                        section selection=w;
                                }
                                amode far_data {
                                        section selection=w;
                                        heap;
                                        stack;
                                }
                        }
                }
        }
}
```

The layout definition is defined with the syntax:

```
layout {
        // space definitions
}
```

The first thing to notice is the different levels inside the layout definition:

space This level can only occur inside a layout level. There are as much space levels as there are space definitions in the cpu part.

block This level can only occur inside a space level. There are as much block levels as there are mappings defined in the space definition in the cpu part.

cluster This level can only occur inside a block level. There can be multiple clusters inside a block. Their main purpose is to group (code/data) sections. The locator locates each cluster in the specified order.

amode This level can only occur inside a cluster level. An **amode** corresponds to an **amode** definition in the cpu part. Within an **amode** you can specify the order in which data/code sections are located.

The four levels can roughly be divided in two groups. The **space** and **block** definition correspond to address ranges and the **cluster** and **amode** definition correspond to (groups of) sections.

The following paragraphs first introduce the **space** and **block** definition. Then separate paragraphs show how to select certain groups of sections and how this is used in the **cluster** and **amode** definition.

5.4.4 Space Definition

Section 5.3.3 already defined the address translation of a space in the cpu part. In the example in that section, the following space was defined:

```
space address_space {
        mau 8;
        map src=0   size=32k dst=0   bus=address_bus label=rom;
        map src=32k size=32k dst=32k bus=address_bus label=ram;
}
```

For every space defined in the cpu part you have to provide a description in the layout definition.

The space level should be inside the layout definition and can only contain one or more block levels.

The name of the space must correspond to a space definition in the cpu part.

The syntax is:

```
space name {
        // block definitions
}
```

Below is an example of a space definition from the software part:

```
space address_space {
        block rom {
                ....
        }
        block ram {
                ...
        }
}
```

In this example space `address_space` defines two blocks: block `rom` and block `ram`.

5.4.5 Block Definition

With the block description you can set boundaries to the sections based on chip sizes.

A block references a physical area of memory. Selected sections are only allowed within the range of the block description. In effect a block limits the range in which a section can be located.

The physical address range of a block is actually defined in the cpu part by a labeled mapping:

```
space address_space {
        mau 8;
        map src=0   size=32k dst=0   bus = address_bus label = rom; //<--
                                             // --> block name: rom
        map src=32k size=32k dst=32k bus = address_bus label = ram; //<--
                                             // --> block name: ram
}
```

The name of the **block** description must correspond to a label in the **map** definition of a **space** definition in the cpu part. The **block** definition must be inside the **space** definition and can only contain one or more cluster levels.

The syntax is:

block *name* **{**
        // cluster definitions
}

Below is an example of a bus definition from the software part:

```
block rom {
        cluster first_code_clstr {
                ...
        }
        cluster code_clstr {
                ...
        }
}
```

In this example block `rom` defines two clusters: cluster `first_code_clstr` and cluster `code_clstr`.

5.4.6 Selecting Sections

The previous paragraphs explained how the address ranges are defined by block definitions, now it is time to select the sections that should be placed in these blocks. In DELFEE there are two levels in which you can define the order of locating:

1. cluster
2. amode

To define the locating order you need to have some kind of handle to specify a section or a group of sections. DELFEE recognizes the following characteristics of a section:

name of the section
: This is unique to a specific section.

attribute(s) of a section
: The attributes of a section are specified by the assembler or compiler. Possible attributes are defined in Table 5.4.6.1. By selecting an attribute you select a group of sections. The attributes can be grouped to an attribute string, for example: **by1w**.

addressing mode
: All sections have an addressing mode (as defined in the cpu part).

Table 5.4.6.1 Section attributes

attr	Meaning	Description
W	Writable	Must be located in ram
R	Read only	Can be located in rom
X	Execute only	Can be located in rom
Z	Zero page	Must be located in the zero page
Y*num*	Addressing mode	Must be located in addressing mode *num*
A	Absolute	Already located by the assembler
B	Blank	Section must be initialized to '0' (cleared)
F	Not filled	Section is not filled or cleared (scratch)
I	Initialize	Section must be initialized in rom
N	Now	Section is located before normal sections (without N or P)
P	Postponed	Section is located after normal sections (without N or P)

To specify a (group) of sections, DELFEE has the following syntax:

1. select a group on section attribute:

   section selection = *attr***;**

2. select a section by name:

   section *name***;**

3. select a special section:

   heap; //locate heap here
   stack; //locate stack here
   table; //locate copy table here
   copy; //locate all initial data here
   copy *name***;** //locate initial data of the named section here

4. create a section:

   reserved label = *name* **length =** *number***;**

Instead of selecting a section by an attribute, DELFEE also allows excluding a section by its attribute.

Excluding an attribute is done by placing a '**-**' (minus sign) in front of *attr*.

So, the example:

```
section selection=attr1-attr2
```

selects a group of sections with attribute *attr1* and without attribute *attr2*.

5.4.7 Cluster Definition

Clusters are used to place specified sections in a group. The locator will handle the clusters in the order that they are specified. This gives you the possibility to create a group of selected sections and give it a higher locate priority.

There are several possibilities to specify that a section is part of a cluster. The exact rules and their priorities are given in Section 5.4.10, "Section Placing Algorithm". The three main possibilities are:

1. attribute
2. section selection=
3. amode definition

Examine the following example:

```
layout {
        space address_space {
                block rom {
                        cluster first_code_clstr {
                                attribute i;
                                amode near_code;
                                amode far_code;
                        }
                        cluster code_clstr {
                                attribute r;
                                amode near_code {
                                        section selection=x;
                                        section selection=r;
                                }
                                amode far_code {
                                        table;
                                        section selection=x;
                                        section selection=r;
                                        copy;      // locate rom copies here
                                }
                        }
                }
        }
}
```

In this example an extra cluster `first_code_cluster` was created. Using the placing algorithm (Section 5.4.10) you can see that sections with attribute 'i' will be placed in cluster `first_code_clstr` and therefore will get a higher priority than sections in cluster `code_clstr`.

The syntax is:

cluster *name* **{**
        // section selections
}

Within a cluster the sections with the least freedom are located first. Freedom is defined by the possible addresses a section can be located at.

5.4.8 Amode Definition

Within a cluster you can specify an addressing mode or amode. Although in the cpu part (Section 5.3.4) an address range was assigned to every amode, in the layout part the addressing mode is used to identify groups of sections.

The syntax is:

```
            :
amode  name {
            section selection = attr;
                        :
}
            :
```

The order of locating is now determined by the order of specification.

For example, suppose you want to locate all writable sections first, then the heap, followed by the stack. In the DELFEE language this is specified by:

```
   :
section selection = w;   // 'w' means writable sections
heap;
stack;
   :
```

5.4.9 Manipulating Sections in Amodes

The previous paragraphs explained how to set the order of the sections within an **amode** definition. DELFEE recognizes an extra set of keywords to further tune the locating of code and data sections.

An **amode** definition can contain the following keywords:

Keyword	Description
section	Selects a section, or group of sections
selection	Specifies attributes for grouping sections
attribute	Assigns attributes (are past to the cluster)
copy	Selects a rom copy of a section by name, or all rom copies in general
fixed	Forces a section to be located around a fixed address
gap	Creates a gap in the address range where sections will not be located
reserved	Reserves a memory area, which can be referenced using locator labels
heap	Defines the place and attributes of the heap
stack	Defines the place and attributes of the stack
table	Defines the place and attributes of the copy table
assert	A user defined assertion
length	Specifies the length of stack, heap, physical block or reserved space

All keywords are described in Section 5.6, "Delfee Keyword Reference".

5.4.10 Section Placing Algorithm

There are different ways to reference a section. Sections can be referenced as a group based on a certain attribute, or they can be referenced very specific by name. To find out where sections are placed in the layout part, DELFEE uses the following algorithm:

1. First, try to find a selection by section name.
2. If not found, search for a 'section selection=' within a matching amode block.
3. If not found, search for a 'section selection=' not within an amode block.
4. If not found, search for a cluster with a correct 'amode= ..,..,.. ;' and correct attributes.
5. If not found, search for a cluster with correct attributes.
6. If not found, relax attribute checking, and start over again.

Relax attributes using the following rules:

1. If stack, heap or reserved, switch indication off and try again.
2. If attribute 'f' (not filled), switch 'f' off and try again.
3. If attribute 'b' (clear), switch 'b' off and try again.
4. If attribute 'i' (initialize), switch 'i' off and try again.
5. If attribute 'x' (executable code), switch 'x' off and 'r' (read-only) on and try again. (Try to place executable sections in read-only memory.)
6. If attribute 'r' (read-only), switch 'r' off 'w' (writable) on and try again. (Try to place read-only sections in writable memory.)

5.5 Memory Part

5.5.1 Introduction

The memory part defines the variable part of the memory configuration. It can be placed in a different file, which allows to easily switch between different memory configurations. The syntax used for the mappings is the same as used in the cpu part.

As you have seen in the example of the cpu part in Section 5.3, there were two references to external busses:

```
bus address_bus {
        mau 8;
        mem addr=0   chips=rom_chip;
        map src=0x100 size=0x7f00  dst=0x100 bus=external_rom_bus;
        mem addr=32k chips=ram_chip;
        map src=0x8100 size=0x7f00 dst=0x100 bus=external_ram_bus;
}
```

In the memory part you have to define the description for the busses `external_rom_bus` and `external_ram_bus`. Using the description in Sections 5.3.5 and 5.3.6 for specifying busses and chips, the memory part could look like:

```
memory {
        bus external_rom_bus {
                mau 8;
                mem addr=0 chips=xrom;
        }

        chips xrom attr=r mau =8 size=0x8000;

        bus external_ram_bus {
                mau 8;
                mem addr=0 chips=xram;
        }

        chips xram attr=w mau=8 size=0x8000;
}
```

5.6 Delfee Keyword Reference

This section contains an alphabetical description of all keywords that can be used in a description file. Some keywords can be abbreviated to a minimum of four characters.

.addr

Syntax:

.addr (Software part)

Description:

The predefined label **.addr** contains the current address.

Example:

```
block ram {
        cluster data_clstr {
                attribute w;
                amode near_data {
                        section selection=w;
                        assert ( .addr < 256, "page overflow");
                                // if the condition is false,
                                // the locator generates an error with
                                // the text as message
                }
                ...
        }
}
```

address

Syntax:

address = *address* (all parts)
addr = *address* (abbreviated form)

Description:

Specify an absolute address in memory.

Example:

Cpu or memory part:

```
bus address_bus {
        mau 8;
        mem addr=0   chips=rom_chip;
        ...
        mem addr=32k chips=ram_chip;
        ...
}
```

Software part:

```
block rom {
        ...
        cluster code_clstr {
                attribute r;
                amode near_code {
                        section selection=x;
                        section selection=r;
                        section .string address = 0x0100;
                }
                ...
        }
}
```

Note that the locate order in the **amode** definition in the example above is fixed. Sections with attribute selection 'x' and/or 'r' are forced to be located before section `.string`. If this fixed order is not desired, the absolute address specification can be done in a separate **amode** definition.
Example:

```
amode near_code {
        section .string address = 0x0100;
}

amode near_code {
        section selection=x;
        section selection=r;
}
```

amode

Syntax:

amode *identifier*[**,** *identifier*]... **{** *amod_description* **}**	(def)	(Cpu or memory part)
amode = *identifier*	(ref)	
amode *identifier*[**,** *identifier*]... **;**		(Software part)
amode *identifier*[**,** *identifier*]... **{** *section_blocks* **}**		

Description:

The keyword **amode** can appear in all parts. In the cpu or memory part you can use **amode** to map an addressing mode or register bank on a particular address space (definition). When you specify **amode=**, you map a specific addressing mode on a previously defined addressing mode (reference). The only keywords allowed in an *amod_description* (cpu part) are **attribute**, **map** and **mau**. The keyword **attribute Y***num* uniquely identifies the addressing mode.
In the software part you can use **amode** as part of a cluster definition to change the locating order of sections. See also Section 5.4.10, "Section Placing Algorithm".

Example:

From cpu or memory part:

```
cpu {
  amode near_data {
        attribute Y3;
        mau 8;
        map src=0 size=1k dst=0 amode = far_data;
                // reference
  }
  amode far_data { // definition
        attribute Y4;
        mau 8;
        map src=0 size=32k dst=32k space = address_space;
  }
```

From software part:

```
block ram {
  cluster data_clstr {
        attribute w;
        amode near_data {
          // Sections with addressing mode
          // near_data are located here
          section selection=w;
        }
        amode far_data {
          // Sections with addressing mode
          // far_data and the stack and heap
          // are located here
          section selection=w;
          heap;
          stack;
        }
  }
}
```

assert

Syntax:

assert (*condition* **,** *text* **) ;** (Software part)
asse (*condition* **,** *text* **) ;** (abbreviated form)

Description:

Test condition of virtual address in memory. Generate an error if the assertion fails and give a message with '*text*'. *condition* is specified as one of:

expr1 > *expr2*
expr1 < *expr2*
expr1 == *expr2*
expr1 != *expr2*

expr1 and *expr2* can be any expression or label. The predefined label **.addr** contains the current address.

Example:

```
block ram {
        cluster data_clstr {
                attribute w;
                amode near_data {
                        section selection=w;
                        assert ( .addr < 256, "page overflow");
                                // if the condition is false,
                                // the locator generates an error with
                                // the text as message
                }
                ...
        }
}
```

attribute

Syntax:

attribute *attribute_string* **;** (Software part)
attr *attribute_string* **;** (abbreviated form)
attribute = *attribute_string* (Software part)
attr = *attribute_string* (abbreviated form)

Description:

With **attribute** you can assign attributes to sections, clusters or memory blocks. See also the keyword **selection**.

For sections these attributes are pure supplementary to the standard section attributes. The standard section attributes such as zero page (Y1), blank (B) and executable (X) are set by the compiler (or by the assembler in the case of an assembler program).

With an action attribute after a section (**attr=**), you can set section attributes or you can disable section attributes with the **-** (minus) sign.

The attributes have the following meaning:

num (Section only) Align the section at 2^{num} MAUs.

Y*num* (amode and sections only) Identify addressing mode. Indicate that sections with this attribute should be allocated in this cluster.

r (Memory and clusters) Indicate this is a read-only cluster or read-only memory.

w (Memory and clusters) Indicate this is a writable cluster or writable memory.

s (Memory only) Indicate this is special memory, it must not be located.

x (Clusters/sections only) Indicate that the cluster/section is executable.

g (Clusters/sections only) Indicate that the cluster/section is global (known in a multi-module environment).

b (Clusters/sections only) Indicate that clusters/sections should be cleared before locating.

i (Sections only) Indicate that clusters/sections should be copied from ROM to RAM.

f (Clusters/sections only) Indicate that clusters/sections should not be filled and not cleared. This is called a scratch cluster/section.

Default attributes if the attribute keyword is omitted:

sections: The attributes as generated from the assembler/compiler.

clusters: The attributes as indicated by the underlying memory, thus **r** for rom and **w** for ram.

memory: If no attributes defined, the default is writable (**w**).

Example:

From software part:

```
layout {
   space address_space {
        block rom {
          cluster first_code_clstr {
                attribute i; // set cluster attribute
                amode near_code;
                amode far_code;
          }
        }
        block ram
          cluster ram {
                amode near_data {
                  // Default attribute of cluster
                  // data is 'w', because the
                  // memory is RAM.

                 section selection=w;
                 section selection=b attr=-b;
                  // Sections with attribute b are
                  // are located here, and
                  // attribute 'b' is switched off
                }
                .
          }
          .
        }
   }
}
```

From cpu part:

```
 amode near_data {
        attribute Y3;  //identify code with Y3
        mau 8;
        map src=0 size=1k dst=0 amode = far_data;
 }
 ...

  chips rom_chip  attr=r mau=8 size=0x100;
  chips ram_chip  attr=w mau=8 size=0x100;
        ...
        // memory attributes
```

block

Syntax:

block *identifier* **{** *block_description* **}** (Software part)

Description:

With **block** you define the contents of a physical area of memory. You can make a **block** description for each chip you use. Each block has a symbolic name as previously defined by the keyword **chips**. It is allowed to combine two or more memory chips in one block as long as their total address range is linear, without gaps. The *identifier* indicates that a memory block starts at the specified chip, no matter how many chips are combined.

Example:

```
layout {
        space address_space {
                block ram
                 // Memory block starting at chip ram_chip
                        cluster ram {
                         ...
                        }
                }
        }
}
```

bus

Syntax:

bus *identifier*[**,** *identifier*]... **{** *bus_description* **}** (def) (Cpu or memory part)
bus = *identifier* (ref)

Description:

With **bus** you define the physical memory addresses for the chips that are located on the cpu (definition). When you specify **bus=**, you map a specific address range on a previously defined address bus (reference). The only keywords allowed in an **bus** description are **mem**, **map** and **mau**.

Example:

```
cpu {
        space address_space {
                // Specify space 'address_space' for the address_bus
                // address bus.
                mau 8;
                map src=0   size=32k dst=0   bus = address_bus label = rom;
                map src=32k size=32k dst=32k bus = address_bus label = ram;
                                                              // ref
        }

        bus address_bus {  // definition
                mau 8;
                mem addr=0   chips=rom_chip;
                map src=0x100 size=0x7f00  dst=0x100 bus = external_rom_bus;
                mem addr=32k chips=ram_chip;
                map src=0x8100 size=0x7f00 dst=0x100 bus = external_ram_bus;
        }
        ...
}
```

chips

Syntax:

chips *identifier*[**,** *identifier*]... *chips_description* (def) (Cpu or memory part)
chips = *identifier*[| *identifier*]... [**,** *identifier*[| *identifier*]...]... (ref)

Description:

With **chips** you describe the chips on the cpu or on your target board (definition). For each chip its **size** and minimum addressable unit (**mau**) is specified. With the keyword **attr** you can define if the memory is read-only. The only three attributes allowed are **r** for read-only, **w** for writable, or **s** for special. If omitted, **w** is default.
You can use **chips=** after the keyword **mem** to specify where a chip is located (reference). You can create chip pairs by separating each chip with a vertical bar '|'.

Example:

```
cpu {
        bus address_bus {
                mau 8;
                mem addr=0 chips=rom_chip; // ref
                ...
        }
        chips rom_chip  attr=r mau=8 size=0x100;  // def
        chips ram_chip  attr=w mau=8 size=0x100;
        ...
}
```

cluster

Syntax:

cluster *cluster_name* **{** *cluster_description* **}** (Software part)
cluster *cluster_name*[**,** *cluster_name*]... **;**

Description:

In the software layout part you can define the cluster name and cluster location order. The attributes as valid for clusters (see **attribute**) can be specified in the first syntax. If you do not specify any attribute, the default attribute **r** or **w** is automatically set.
In a cluster description you can not only determine the locate order of sections within the named cluster, but you can also specify stack and heap size, extra process memory, define labels for the process, etc.

Example:

```
space address_space {
        block rom {
                cluster first_code_clstr {
                 // The default attribute 'r' of cluster
                 // text is overruled to  'i'. All sections with attribute
                 // 'i' are located here by default.
                        attribute i;
                        amode near_code;
                        amode far_code;
                         // Sections with addressing mode
                         // near_code or far_cdoe are located here
                }
        block ram {
                cluster data_clstr {
                 // default attribute 'w' because the memory is RAM.
                 // All writable sections are located here by default.
                        attribute w; // can be omitted
                        amode near_data {
                                section selection=w;
                        }
        }
}
```

copy

Syntax:

copy *section_name* [**attr** = *attribute*] **;** (Software part)
copy selection = *attribute* [**attr** = *attribute*] **;**
copy ;

Description:

The ROM copy of data sections with the attribute **i** will be copied from ROM to RAM at program startup. With **copy** you define the placement in memory of these ROM copies. You can specify a specific section by giving the section's name, or select sections with a specific attribute. If you do not specify an argument, the locator locates all ROM copies at the specified location. With **attr=** you can change the section attributes.

If you do not specify the keyword **copy** at all, the locator finds a suitable place for ROM copies.

See also the keywords **attribute** and **selection**.

Example:

```
space address_space {
  block rom {
        ...
        cluster code_clstr {
                attribute r; //cluster attribute
                amode far_code {
                  table;
                  section selection=x;
                  section selection=r;
                  copy; // all ROM copies are located here
                }
        }
}
```

cpu

Syntax:

cpu { *cpu_description* } (Cpu part)
cpu *filename*

Description:

The keyword **cpu** appears together with **software** and **memory** at the highest level in a description file. The actual cpu description starts between the curly braces **{ }**. Normally you do not need to change the cpu part because it is delivered with the product and describes the derivative completely.

The second syntax is the so-called include syntax. The locator opens the file *filename* and reads the actual cpu description from this file. You must start the included file with **cpu** again. The *filename* can contain a complete path including a drive letter. Parts of *filename*, or the complete *filename* can be put in a environment variable. The file is first searched for in the current directory, and secondly in the `etc` directory relative to the installation directory.

Example:

Contents of the description file:

```
software {
        ...
}

cpu target.cpu  //cpu part in separate file
memory target.mem
```

See Section 5.3 for a sample contents of a `.cpu` file.

dst

Syntax:

dst = *address* (Cpu or memory part)

Description:

Specify destination address as part of the keyword **map** in an **amode**, **space** or **bus** description. For *address* you can use any decimal, hexadecimal or octal number. You can also use the (standard) Delfee suffix **k**, for kilo (2^{10}) or **M**, for mega (2^{20}). The unit of measure depends on the MAU (minimum addressable unit) of the destination memory space.

Example:

```
cpu {
        ...
        amode near_code {
                attribute Y1;
                mau 8; // 8-bit addressable
                map src=0 size=1k dst=0 amode=far_code;
        }
}
```

fixed

Syntax:

fixed address = *address* ; (Software part)
fixed addr = *address* ; (abbreviated form)

Description:

Define a fixed point in the memory map. The locator allocates the section/cluster preceding the fixed definition and the section/cluster following it as close as possible to the fixed point.

Example:

```
block ram {
        cluster near_data_clstr {
          amode near_data {
                section selection=w;
                fixed  addr = 0x2000;
          }
        }
        cluster far_data_clstr;
}
```

Cluster `far_data_clstr` will be located with its upper bound at address 0x2000 and cluster `near_data_clstr` starts at this address. The same can be applied to sections.

gap

Syntax:

gap; (Software part)
gap length = *value* ;

Description:

Reserve a gap with a dynamic size. The locator tries to make the memory space as big as possible. You can use this keyword in a block description to create a gap between clusters, or in a cluster description to create a gap between sections. You can also use the **gap** keyword in combination with the **fixed** keyword.

With the second form you can specify a gap of a fixed length. This form can only occur in a block description.

Example:

```
space address_space {
        block ram {
                cluster data_clstr {
                        attr w;
                        amode near_data;
                } // low side mapping

                gap;    // balloon
                cluster stck; // high side mapping
        }
}
```

heap

Syntax:

heap *heap_description* **;** (Software part)
heap ;

Description:

Like **table** and **stack**, **heap** is another special section. The section is not created from the `.out` file, but generated at locate time. To control the size of this special section the keyword **length** is allowed within the heap description. You can use **heap** to include dynamic memory for a process.

Heap can only be used if a malloc() function has been implemented.

Two locator labels are used to mark begin and end of the heap, `__lc_bh` for the begin of heap, and `__lc_eh` for the end of heap.

Note that if the **heap** keyword is specified in the description file this does not automatically mean that a heap will always be generated. A heap will only be allocated when its section labels (`__lc_bh` for begin of heap and `__lc_eh` for end of heap) are used in the program.

The heap description can be a length specification and/or an attribute specification. See the example.

Example:

```
layout {
        space address_space {
                block ram {
                        cluster data_clstr {
                                amode far_data {
                                        section selection=w;
                                        heap length=100;
                                                // Heap of 100 MAUs
                                }
                        }
                }
        }
}
```

label

Syntax:

label *identifier* **;** (Software part)
label = *identifier* **;** (All parts)

Description:

The first form can be used stand-alone to specify a virtual address in memory by means of a label. The virtual address is label **__lc_u_***identifier*. Note that at C level, all locator labels start with one underscore (the compiler adds another underscore '_').

The second form can only be used as part of another keyword. As part of the keyword **reserved** you can assign a label to an address range. The start of the address range is identified by label **__lc_ub_***identifier*. The end of the address range is identified by label **__lc_ue_***identifier*. The keyword **label** is also allowed as part of the **map** keyword to assign a name to a block of memory in a space definition.

Example:

From the software part:

```
block ram {
        cluster data_clstr {
                attribute w;
                amode far_data {
                        section selection=w;
                        heap;
                        stack;
                        reserved label=xvwbuffer length=0x10;
                        // Start address of reserved area is
                        // label __lc_ub_xvwbuffer
                        // End address of reserved area is
                        // label __lc_ue_xvwbuffer
                }
        }
}
```

From the cpu part:

```
space address_space {
        mau 8;
        map src=0   size=32k dst=0   bus = address_bus label=rom;
        map src=32k size=32k dst=32k bus = address_bus label=ram;
}
```

layout

Syntax:

layout { *layout_description* **}** (Software part)
layout *filename*

Description:

The **layout** part describes the layout of sections in memory. The **layout** part groups sections into clusters and you can define the name, number and the order of clusters. The **layout** part describes how these clusters must be allocated into physical RAM and ROM block. The space and block names used in the **layout** part must be present in the memory part or the cpu part. The cluster definitions can contain fixed addresses as well as definitions of gaps between sections.

Example:

```
software {
  layout {
    space address_space {
        block rom {
          cluster first_code_clstr {
            attribute i;
            amode near_code;
          }
        ....
```

length

Syntax:

length = *length* (Cpu, memory and software part)
leng = *length* (abbreviated form)

Description:

You can use the keyword **length** to define the length in MAUs (minimum addressable units) of a certain memory area. *length* must be a numeric value and can be given either in hex, octal or decimal. As usual, hex numbers must start with '0x' and octal numbers must start with '0'. You can use the suffix **k** which stands for kilo or **M** which stands for mega.

You can use **length** to specify the length of the reserved memory or to specify the stack, heap or gap length. For details see the keywords **reserved**, **stack**, **heap** and **gap**.

Example:

```
space address_space {
        block ram {
                cluster data_clstr {
                        amode far_data {
                                stack leng = 2k;
                        }
                }
        }
}
```

load_mod

Syntax:

load_mod *identifier* **start** = *label*; (Software part)
load_mod start = *label*;

Description:

With **load_mod** you are introducing a load module description. This keyword is followed by an optional identifier, representing a load module name with or without the `.out` extension. The load module itself must be supplied to the locator as a parameter in the invocation. If the identifier is omitted, the load module is taken from the command line.

Example:

```
software {
        load_mod start = __START;
}
```

or

```
software {
        load_mod hello start = __USER_start;
}
```

map

Syntax:

map *map_description* (Cpu or memory part)

Description:

Map a memory part, specified as a source address and a size, to a destination address of an **amode**, **space** or **bus**. The unit of measure depends on the MAU of the memory space.

Example:

```
cpu {
  .
  amode far_data {
        attribute Y4;
        mau 8;
        map src=0 size=32k dst=32k space=address_space;
  }
  space address_space {
        mau 8;
        map src=0   size=32k dst=0   bus = address_bus label=rom;
        map src=32k size=32k dst=32k bus = address_bus label=ram;
  }
  bus address_bus {
        mau 8;
        mem addr=0   chips=rom_chip;
        map src=0x100 size=0x7f00  dst=0x100 bus=external_rom_bus;
        mem addr=32k chips=ram_chip;
        map src=0x8100 size=0x7f00 dst=0x100 bus=external_ram_bus;
  }
  .
}
```

mau

Syntax:

mau *number* ; (Cpu or memory part)

mau = *number*

Description:

You can use the keyword **mau** to specify the minimum addressable unit in bits of a certain memory area. The first form can only be used in an **amode**, **space** or **bus** description. The second form can be used to specify the minimum addressable unit of a chip. Note that **mau** affects the unit of measure for other keywords. If no **mau** is specified, the default number is 8 (byte addressable).

Example:

```
cpu {
  amode near_code {
     attribute Y1;
     mau 8;     // byte addressable
     map src=0 size=1k dst=0 amode=far_code;
               // src is at address 0,
               // size is 1k byte units
               // dst is at address 0
  }
}
```

mem

Syntax:

mem *mem_description* ; (Cpu or memory part)

Description:

Define the start address of a chip in memory. The only keywords allowed in a mem description are **address** and **chips**.

Example:

```
cpu {
  ...
  bus internal_bus {
    mau 8;
    mem addr=0    chips=rom_chip;
        // chip 'rom_chip' is located at memory
        // address 0
        ...
    mem addr=32k  chips=ram_chip;
        // chip 'ram_chip' is located at memory
        // address 0x8000
        ...

  }
  chips rom_chip  attr=r mau=8 size=0x100;
  chips ram_chip  attr=w mau=8 size=0x100;

}
```

memory

Syntax:

memory { *memory_description* } (Memory part)
memory *filename*

Description:

Together with **software** and **cpu**, **memory** introduces a main part of the description file. You can specify the actual memory part between the curly braces **{ }**.

You can use the memory part to describe any additional memory or addresses of peripherals not integrated on the cpu.

The second syntax is the include syntax. In this case, the memory part is defined in a separate file. This included file must start again with **memory**. The *filename* can contain a complete path, including a drive letter. You can put parts of *filename*, or the complete *filename* in an environment variable. The file is first searched for in the current directory, and secondly in the `etc` directory relative to the installation directory.

Example:

```
software {
        ...
}

cpu target.cpu
memory target.mem       //mem part in separate file
```

See Section 5.5 for a sample contents of a `.mem` file.

regsfr

Syntax:

regsfr *filename* (Cpu or memory part)

Description:

Specify a register file generated by the register manager for use by the debugger.

Example:

```
cpu {
        .
        .
        regsfr regfile.dat
        /*
         * Use file regfile.dat generated by register manager
         */
}
```

reserved

Syntax:

reserved *reserved_description* **;** (Software part)
reserved ;

Description:

Reserve a fixed amount of memory space or reserve as much memory as possible in the memory space. If no length is specified the size of the memory allocation depends on the size of the memory space or the size is limited by a fixed point definition following the **reserved** allocation.

You can only use the keywords **address**, **attribute**, **label** and **length** in the reserved description. You can use the keyword **reserved** in an amode description.

Example:

```
space address_space {
        block rom {
                cluster code_clstr {
                        amode near_code {
                          // system reserved
                          // (exception vector)
                          reserved length=0x2 addr=0x24;
                        }
        }
}
```

section

Syntax:

section *identifier* [**addr =** *address*] [**attr =** *attribute*] **;** (Software part)
section selection = *attribute* [**addr =** *address*] [**attr =** *attribute*] **;**

Description:

section can be used in the layout part to specify the location order within a cluster. See also **layout**. The *identifier* is the name of a section.

With **addr=** you can make a section absolute.
With **attr=** you can assign new attributes to a section or disable attributes.

See also the keywords **address**, **attribute** and **selection**.

Example:

```
space address_space {
        block ram {
                cluster data_clstr {
                        amode near_data {
                          // locate section .data here and set
                          // attribute 'w'
                          section .data attr=w;
                          section selection=b attr=-b;
                        }
                }
        }
}
```

selection

Syntax:

selection = *attribute*

Description:

You can use **selection** after the keywords **section** or **copy** to select all sections with (a) specified attribute(s).

If more attributes are specified, only sections with all attributes are selected. If a minus sign '**-**' precedes the attribute, only sections **not** having the attribute are selected.

See also the keywords **attribute**, **copy** and **section**.

Example:

```
space address_space {
        block ram {
          cluster data_clstr {
                amode near_data {
                 // select sections with w on and not i.
                 // (select all writable sections which
                 // are not copied from ROM)
                 section selection=-iw;
                }
          }
        }
        .
}
...
```

size

Syntax:

size = *size* (Cpu or memory part)

Description:

You can use the keyword **size** to define the size in minimum addressable units (MAU) of a certain memory area. *size* must be a numeric value and can be given either in hex, octal or decimal. As usual, hex numbers must start with '0x' and octal numbers must start with '0'. You can use the suffix **k** which stands for kilo or **M** which stands for mega.

You can use **size** to specify the size of a part of memory that must be mapped on another part of memory or to specify the size of a chip. For details see the keywords **map** and **chips**.

Example:

```
cpu {
  amode near_code {
        attribute Y1;  //identify near_code with Y1
        map src=0 size=1k dst=0 amode=far_code;
  }
  space address_space {
        mau 8;
        map src=0   size=32k dst=0   bus=address_bus label=rom;
        map src=32k size=32k dst=32k bus=address_bus label=ram;
  }
  chips rom_chip attr=r mau=8 size=0x100;
  chips ram_chip attr=w mau=8 size=0x100;
        // size of chips
}
```

software

Syntax:

software { *software_description* } (Software part)
software *filename*

Description:

The keyword **software** appears at the highest level in a description file. The actual software description starts between the curly braces **{ }**.
The second syntax is the so called include syntax. The locator will open file *filename* and read the actual software description from this file. The first keyword in *filename* must be **software** again. The *filename* can contain a complete path including a drive letter. You can put parts of *filename*, or the complete *filename* in an environment variable. The file is first searched for in the current directory, and secondly in the `etc` directory relative to the installation directory.

Example:

Contents of the description file:

```
software $(MY_OWN_DESCRIPTION)

cpu target.cpu
memory target.mem
```

Environment variable `MY_OWN_DESCRIPTION` contains the name of a file with contents like:

```
software {
        load_mod start = __START;
        layout {
        .
        .
        }
}
```

space

Syntax:

space *identifier* **{** *space_description* **}** (Software part)
space *identifier*[**,** *identifier*]... **{** *space_description* **}** (Cpu or memory part)
space = *identifier*

Description:

The keyword **space** can be used in the cpu part, memory part and software part. In the cpu or memory part you can use **space** to describe a physical memory address space. The only keywords allowed in a space description in the cpu or memory part are **mau** and **map**.
In the software part you can use **space** to describe one or more memory blocks. Each space has a symbolic name as previously defined by the keyword **space** in the cpu or memory part.

Example:

From the cpu part:

```
cpu {
  amode far_data {
        attribute Y4;
        mau 8;
        map src=0 size=32k dst=32k space=address_space;
  }
        ...
  space address_space {
        // Specify space 'address_space' for the
        // address_bus address bus.
        mau 8;
        map src=0   size=32k dst=0   bus=address_bus label=rom;
        map src=32k size=32k dst=32k bus=address_bus label=ram;
  }
  .
}
```

From the software part:

```
layout {
        // define the preferred locating order of sections
        // in the memory space
        // (the range is defined in the .cpu file)
        space address_space {
        ...

                // define for each sub-area in the space
                // the locating order of sections
                block rom {
                  // Memory block starting at chip rom_chip

                  // define a cluster for read-only sections
                  cluster code_clstr {
                        ....
                  }
                }
        .
        }
}
```

src

Syntax:

src = *address* (Cpu or memory part)

Description:

Specify source address as part of the keyword **map** in an amode, space or bus description. For *address* you can use any decimal, hexadecimal or octal number. You can also use the (standard) Delfee suffix **k**, for kilo (2^{10}) or **M**, for mega (2^{20}). The address is specified in the addressing mode's local MAU (minimum addressable unit) size (default 8 bits).

Example:

```
cpu {
  ...
  amode near_code {
        attribute Y1;
        mau 8; // 8-bit addressable
        map src=0 size=1k dst=0 amode=far_code;
  }
}
```

stack

Syntax:

stack *stack_description* **;** (Software part)

stack ;

Description:

stack is a special form of a section description. The stack is allocated at locate time. The locator only allocates a stack if one is needed. Two special locator labels are associated with the stack space located with keyword **stack**. The begin of the stack area can be obtained by the locator label `__lc_bs`, the end address is accessible by means of label `__lc_es`.

If the stack grows downwards the begin of stack must be the highest address. To accomplish this, you can keep the length positive and set the stack pointer to *end_of_stack*, so the formula:

end_of_stack = *begin_of_stack* + length

is always true.

You can only use the keywords **attribute** and **length** in the stack description. If you specify **stack** without a description, the locator tries to make the stack as big as possible. If you do not specify the keyword **stack** at all, the locator also tries to make the stack as big as possible but at least 100 (MAUs).

Example:

```
space address_space {
        block ram {
                cluster data_clstr {
                        amode far_data {
                                section selection=w;
                                stack leng=150;
                                // stack of 150 MAUs
                                ...
                        }
                }
        }
}
```

start

Syntax:

start = *label* **;** (Software part)

Description:

Define a start label for a process.

You can use **start** only within a load module description.

Example:

```
software {
        load_mod start = system_start;

        layout {
        .
        .
        }
}
```

table

Syntax:

table attr = *attribute* **;** (Software part)
table ;

Description:

Like **stack** and **heap** also **table** is a special kind of section. Normal sections are generated at compile time, and passed via the assembler and linker to the locator. The stack and heap sections are generated at locate time, with a user requested size.

table is different. The locator is able to generate a copy table. Normally, this table is put in read-only memory. If you want to steer the table location, you can use the **table** keyword. With table only **attribute** is allowed. The length is calculated at locate time. **table** can occur in a cluster description.

Example:

```
space address_space {
  block rom {
        ...
        cluster code_clstr {
                attribute r; // cluster attribute
                amode far_code {
                  table; // locate copy table here
                  section selection=x;
                  section selection=r;
                  copy; // all ROM copies are located here
                }
        }
}
```

5.6.1 Abbreviation of Delfee Keywords

The following Delfee keywords can be abbreviated to unique 4 character words:

Table 5.6.1.1 Abbreviation of Delfee keywords

Keyword	Abbreviation
address	addr
assert	asse
attribute	attr
length	leng

5.6.2 Delfee Keywords Summary

Table 5.6.2.1 Overview of Delfee keywords

Keyword	Description
address	Specify absolute memory address
amode	Specify the addressing modes
assert	Error if assertion failed
attribute	Assign attributes to clusters, sections, stack or heap
block	Define physical memory area
bus	Specify address bus
chips	Specify cpu chips
cluster	Specify the order and placement of clusters
copy	Define placement of ROM-copies of data sections
cpu	Define cpu part
dst	Destination address
fixed	Define fixed point in memory map
gap	Reserve dynamic memory gap
heap	Define heap
label	Define virtual address label
layout	Start of the layout description
length	Length of stack, heap, physical block or reserved space
load_mod	Define load module (process)
map	Map a source address on a destination address
mau	Define minimum addressable unit (in bits)
mem	Define physical start address of a chip
memory	Define memory part
regsfr	Specify register file for use by debugger
reserved	Reserve memory
section	Define how a section must be located
selection	Specify attributes for grouping sections into clusters
size	Size of address space or memory
software	Define the software part
space	Define an addressing space or specify memory blocks
src	Source address
stack	Define a stack section
start	Give an alternative start label
table	Define a table section

CHAPTER 6 UTILITIES

6.1 Overview

The following utilities are supplied with the Cross-Assembler for the S1C processor family which can be useful at various stages during program development.

ar88 An IEEE archiver. This is a librarian facility, which can be used to create and maintain object libraries.

cc88 A control program for the S1C tool chain.

mk88 A utility program to maintain, update, and reconstruct groups of programs.

pr88 An IEEE object reader that views the contents of files which have been created by a tool from the S1C tool chain.

The utilities are explained on the following pages.

6.2 ar88

Name

ar88 IEEE archiver and library maintainer

Synopsis

ar88 *key_option* [*option*]... *library* [*object_file*]...
ar88 **-V**
ar88 **-?**

Description

With **ar88** you can combine separate object modules in a library file. The linker optionally includes modules from a library when a specific module resolves an external symbol definition in one of the modules that has been read before. The library maintainer **ar88** is a program to build library files and it offers the possibility to replace, extract or remove modules from an existing library.

key_option one of the main options indicating the action **ar88** has to take. Key options may appear in any order, at any place.

option optional sub-options as explained on the next pages.

library is the library file.

object_file is an object module to be added, extracted, replaced or removed from the library.

Options

You may specify options with or without a leading '**-**'. Options may occur in random order. You may also combine options. So **-xv** is allowed. **-V** and **-?** however, must be the only option on the command line.

Key options:

-d

Delete the named object modules from the library.

-m

Move the named object modules to the end of the library, or to another position as specified by one of the positioning options.

-p

Print the named object modules in the library on standard output.

Note: the object is in binary format. The option is normally used with a redirection:

```
ar88 -p lib.a object.obj > t.obj
```

-r

Replace the named object modules in the library if they exist. If they are not in the library, add them. If no names are given, only those object modules are replaced for which a file with the same name is found in the current directory. New modules are placed at the end.

-t

Print a table of contents of the library. If no names are given, all object modules in the library are printed. If names are given, only those object modules are tabled.

-x

Extract the named object modules from the library. If no names are given, all modules are extracted from the library. In neither case does **x** alter the library.

Other options:

-?

Display an explanation of options at `stdout`.

-V

Display version information at `stderr`.

-a *posname*
Append or move new object modules after existing module *posname*. This option can only be used in combination with the **m** or **r** option.

-b *posname*
Insert or move new object modules before existing module *posname*. This option can only be used in combination with the **m** or **r** option.

-c
Create the library file without notification if the library does not exist.

-f *file*
Read options from file *file*. '**-**' means `stdin`. You need to provide the EOF code to close `stdin` (usually Ctrl-Z).

-o
Reset the last-modified date to the date recorded in the library. It can only be used in combination with the **x** option.

-s
Print a list of symbols. This option must be combined with **-t**.

-s1
Print a list of symbols. Each symbol is preceded by the library name and the name of the object file. This option must be combined with **-t**.

-u
Replace only those object modules with the last-modified date later than the library file. It can only be used in combination with the **r** option.

-v
Verbose. Under the verbose option, **ar88** gives a module-by-module description of the making of a new library file from the old library and the constituent modules. It can only be used in combination with the **d**, **m**, **r**, or **x** option.

-w*n*
Set warning level *n*.

Examples

1. Create library **clib.a** consisting of the modules **startup.obj**, and **calc.obj**:

```
ar88 cr clib.a startup.obj calc.obj
```

2. Extract all modules form library **clib.a**:

```
ar88 x clib.a
```

3. Print a list of symbols from library **clib.a**:

```
ar88 ts clib.a
startup.obj
  symbols:
        _start
        _copytable
calc.obj
  symbols:
       _entry
```

4. Print a list of symbols from library **clib.a** in a different form:

```
ar88 ts1 clib.a
clib.a:startup.obj:_start
clib.a:startup.obj:_copytable
clib.a:calc.obj:_entry
```

5. Delete module **calc.obj** from library **clib.lib**:

```
ar88 d clib.a calc.obj
```

6.3 cc88

Name

cc88 control program for the S1C tool chain

Synopsis

cc88 [[*option*]... [*control*] ... [*file*]...]...
cc88 **-V**
cc88 **-?**

Description

The control program **cc88** facilitates the invocation of the various components of the S1C family tool chain from a single command line. The control program accepts source files and options on the command line in random order.

Options are preceded by a '**-**' (minus sign). The input *file* can have one of the extensions explained below.

The control program recognizes the following argument types:

- Arguments starting with a '**-**' character are options. Some options are interpreted by the control program itself; the remaining options are passed to those programs in the tool chain that accept the option.
- Arguments with a `.c` suffix are interpreted as C source programs and are passed to the compiler.
- Arguments with a `.asm` suffix are interpreted as assembly source files which have to be preprocessed and passed to the assembler.
- Arguments with a `.src` suffix are interpreted as compiled assembly source files. They are directly passed to the assembler.
- Arguments with a `.a` suffix are interpreted as library files and are passed to the linker.
- Arguments with a `.obj` suffix are interpreted as object files and are passed to the linker.
- Arguments with a `.out` suffix are interpreted as linked object files and are passed to the locator. The locator accepts only one `.out` file in the invocation.
- Arguments with a `.dsc` suffix are treated as locator command files. If there is a file with extension `.dsc` on the command line, the control program assumes a locate phase has to be added. If there is no file with extension `.dsc`, the control program stops after linking (unless it has been directed to stop in an earlier phase)
- If other arguments are found, an error message is given.

Normally, a control program tries to compile and assemble all source files to object files, followed by a link and locate phase which produces an absolute output file. There are however, options to suppress the assembler, linker or locator stage. The control program produces unique filenames for intermediate steps in the compilation process, which are removed afterwards. If the compiler and assembler are called subsequently, the control program prevents preprocessing of the compiler generated assembly file. Normally, assembly input files are preprocessed first.

Options

-?

Display a short explanation of options at `stdout`.

-M{**s**|**c**|**d**|**l**}

Specify the memory model to be used:

small	(**s**)
compact data	(**d**)
compact code	(**c**)
large	(**l**)

-V

The copyright header containing the version number is displayed, after which the control program terminates.

-Ta *arg* / **-Tc** *arg* / **-Tlk** *arg* / **-Tlc** *arg*

With these options you can pass a command line argument directly to the assembler (**-Ta**), C compiler (**-Tc**), linker (**-Tlk**) or locator (**-Tlc**). These options may be used to pass some options that are not recognized by the control program, to the appropriate program. The argument may be either directly appended to the option, or follow the option as a separate argument of the control program.

-al

Generate an absolute list file for each module in the application.

-c / **-cl** / **-cs**

Normally, the control program invokes all stages to build an absolute file from the given input files. With these options it is possible to skip the C compiler, assembler, linker or locator stage. With the **-cs** option the control program stops after the compilation of the C source files (`.c`) and after preprocessing the assembly source files (`.asm`), and retains the resulting `.src` files. With the **-c** option the control program stops after the assembler, with as output one or more object files (`.obj`). With the **-cl** option the control program stops after the link stage, with as output a linker object file (`.out`).

-f *file*

Read command line arguments from *file*. The filename "**-**" may be used to denote standard input. To get around the limits on the size of the command line, it is possible to use command files. These command files contain the options that could not be part of the real command line. Command files can also be generated on the fly, for example by the make utility.
Some simple rules apply to the format of the command file:

1. It is possible to have multiple arguments on the same line in the command file.
2. To include whitespace in the argument, surround the argument with either single or double quotes.
3. If single or double quotes are to be used inside a quoted argument, we have to go by the following rules:
   a. If the embedded quotes are only single or double quotes, use the opposite quote around the argument. Thus, if a argument should contain a double quote, surround the argument with single quotes.
   b. If both types of quotes are used, we have to split the argument in such a way that each embedded quote is surrounded by the opposite type of quote.

   Example:
   ```
   "This has a single quote ' embedded"
   ```
   or
   ```
   'This has a double quote " embedded'
   ```
   or
   ```
   'This has a double quote " and a single quote '"' embedded"
   ```
4. Some operating systems impose limits on the length of lines within a text file. To circumvent this limitation it is possible to use continuation lines. These lines end with a backslash and newline. In a quoted argument, continuation lines will be appended without stripping any whitespace on the next line. For non-quoted arguments, all whitespace on the next line will be stripped.

   Example:
   ```
   "This is a continuation \
   line"
        → "This is a continuation line"

   control(file1(mode,type),\
   file2(type))
        → control(file1(mode,type),file2(type))
   ```
5. It is possible to nest command line files up to 25 levels.

-ieee / **-srec**

With these options you can specify the locator output format of the absolute file. The output file can be an IEEE-695 file (`.abs`) or Motorola S-record file (`.sre`). The default output is IEEE-695 (`.abs`).

-nolib

With this option the control program does not supply the standard libraries to the linker. Normally the control program supplies the default C and run-time libraries to the linker. Which libraries are needed is derived from the compiler options.

-o *file*

Normally, this option is passed to the locator to specify the output file name. When you use the **-cl** option to suppress the locating phase, the **-o** option is passed to the linker. When you use the **-c** option to suppress the linking phase, the **-o** option is passed to the assembler, provided that only one source file is specified. When you use the **-cs** option to suppress the assembly phase, the **-o** option is passed to the compiler. The argument may be either directly appended to the option, or follow the option as a separate argument of the control program.

-tmp

With this option the control program creates intermediate files in the current directory. They are not removed automatically. Normally, the control program generates temporary files for intermediate translation results, such as compiler generated assembly files, object files and the linker output file. If the next phase in the translation process completes successfully, these intermediate files will be removed.

-v

When you use the **-v** option, the invocations of the individual programs are displayed on standard output, preceded by a '+' character.

-v0

This option has the same effect as the **-v** option, with the exception that only the invocations are displayed, but the programs are not started.

Environment Variables used by cc88

The control program uses the following environment variables:

TMPDIR

This variable may be used to specify a directory, which the control programs should use to create temporary files. When this environment variable is not set, temporary files are created in the current directory.

CC88OPT

This environment variable may be used to pass extra options and/or arguments to each invocation of the control program. The control program processes the arguments from this variable before the command line arguments.

CC88BIN

When this variable is set, the control program prepends the directory specified by this variable to the names of the tools invoked.

6.4 mk88

Name

mk88 maintain, update, and reconstruct groups of programs

Synopsis

mk88 [*option*]... [*target*]... [*macro*=*value*]...
mk88 **-V**
mk88 **-?**

Description

mk88 takes a file of dependencies (a 'makefile') and decides what commands have to be executed to bring the files up-to-date. These commands are either executed directly from **mk88** or written to the standard output without executing them.

If no target is specified on the command line, **mk88** uses the first target defined in the first makefile.

Options

-?
Show invocation syntax.

-D
Display the text of the makefiles as read in.

-DD
Display the text of the makefiles and 'mk88.mk'.

-G *dirname*
Change to the directory specified with *dirname* before reading a makefile. This makes it possible to build an application in another directory than the current working directory.

-S
Undo the effect of the **-k** option. Stop processing when a non-zero exit status is returned by a command.

-V
Display version information at `stderr`.

-W *target*
Execute as if this target has a modification time of "right now". This is the "What If" option.

-d
Display the reasons why **mk88** chooses to rebuild a target. All dependencies which are newer are displayed.

-dd
Display the dependency checks in more detail. Dependencies which are older are displayed as well as newer.

-e
Let environment variables override macro definitions from makefiles. Normally, makefile macros override environment variables. Command line macro definitions always override both environment variables and makefile macros definitions.

-f *file*
Use the specified file instead of 'makefile'. A **-** as the makefile argument denotes the standard input.

-i
Ignore error codes returned by commands. This is equivalent to the special target .IGNORE:.

-k
When a nonzero error status is returned by a command, abandon work on the current target, but continue with other branches that do not depend on this target.

-n

Perform a dry run. Print commands, but do not execute them. Even lines beginning with an @ are printed. However, if a command line is an invocation of **mk88**, that line is always executed.

-q

Question mode. **mk88** returns a zero or non-zero status code, depending on whether or not the target file is up to date.

-r

Do not read in the default file 'mk88.mk'.

-s

Silent mode. Do not print command lines before executing them. This is equivalent to the special target .SILENT:.

-t

Touch the target files, bringing them up to date, rather than performing the rules to reconstruct them.

-w

Redirect warnings and errors to standard output. Without, **mk88** and the commands it executes use standard error for this purpose.

macro=value

Macro definition. This definition remains fixed for the **mk88** invocation. It overrides any regular definitions for the specified macro within the makefiles and from the environment. It is inherited by subordinate **mk88**'s but act as an environment variable for these. That is, depending on the **-e** setting, it may be overridden by a makefile definition.

Usage

Makefiles

The first makefile read is 'mk88.mk', which is looked for at the following places (in this order):

- in the current working directory
- in the directory pointed to by the HOME environment variable
- in the `etc` directory relative to the directory where **mk88** is located

Example:

when **mk88** is installed in `\C88\BIN` the directory `\C88\ETC` is searched for makefiles.

It typically contains predefined macros and implicit rules.

The default name of the makefile is 'makefile' in the current directory. Alternate makefiles can be specified using one or more **-f** options on the command line. Multiple **-f** options act as if all the makefiles were concatenated in a left-to-right order.

The makefile(s) may contain a mixture of comment lines, macro definitions, include lines, and target lines. Lines may be continued across input lines by escaping the NEWLINE with a backslash (\). If a line must end with a backslash then an empty macro should be appended. Anything after a "#" is considered to be a comment, and is stripped from the line, including spaces immediately before the "#". If the "#" is inside a quoted string, it is not treated as a comment. Completely blank lines are ignored.

An *include* line is used to include the text of another makefile. It consists of the word "include" left justified, followed by spaces, and followed by the name of the file that is to be included at this line. Macros in the name of the included file are expanded before the file is included. Include files may be nested.

An *export* line is used for exporting a macro definition to the environment of any command executed by **mk88**. Such a line starts with the word "export", followed by one or more spaces and the name of the macro to be exported. Macros are exported at the moment an export line is read. This implies that references to forward macro definitions are equivalent to undefined macros.

Conditional Processing

Lines containing `ifdef`, `ifndef`, `else` or `endif` are used for conditional processing of the makefile. They are used in the following way:

```
ifdef macroname
if-lines
else
else-lines
endif
```

The *if-lines* and *else-lines* may contain any number of lines or text of any kind, even other `ifdef`, `ifndef`, `else` and `endif` lines, or no lines at all. The `else` line may be omitted, along with the *else-lines* following it.

First the *macroname* after the `if` command is checked for definition. If the macro is defined then the *if-lines* are interpreted and the *else-lines* are discarded (if present). Otherwise the *if-lines* are discarded; and if there is an `else` line, the *else-lines* are interpreted; but if there is no `else` line, then no lines are interpreted.

When using the `ifndef` line instead of `ifdef`, the macro is tested for not being defined. These conditional lines can be nested up to 6 levels deep.

Macros

Macros have the form 'WORD = text and more text'. The WORD need not be uppercase, but this is an accepted standard. Spaces around the equal sign are not significant. Later lines which contain $(WORD) or ${WORD} will have this replaced by 'text and more text'. If the macro name is a single character, the parentheses are optional. Note that the expansion is done recursively, so the body of a macro may contain other macro invocations. The right side of a macro definition is expanded when the macro is actually used, not at the point of definition.

Example:

```
FOOD = $(EAT) and $(DRINK)
EAT = meat and/or vegetables
DRINK = water
export FOOD
```

'$(FOOD)' becomes 'meat and/or vegetables and water' and the environment variable FOOD is set accordingly by the export line. However, when a macro definition contains a direct reference to the macro being defined then those instances are expanded at the point of definition. This is the only case when the right side of a macro definition is (partially) expanded. For example, the line

```
DRINK = $(DRINK) or beer
```

after the export line affects '$(FOOD)' just as the line

```
DRINK = water or beer
```

would do. However, the environment variable FOOD will only be updated when it is exported again.

Special Macros

MAKE

This normally has the value **mk88**. Any line which invokes MAKE temporarily overrides the **-n** option, just for the duration of the one line. This allows nested invocations of MAKE to be tested with the **-n** option.

MAKEFLAGS

This macro has the set of options provided to **mk88** as its value. If this is set as an environment variable, the set of options is processed before any command line options. This macro may be explicitly passed to nested **mk88**'s, but it is also available to these invocations as an environment variable. The **-f** and **-d** flags are not recorded in this macro.

PRODDIR

This macro expands the name of the directory where **mk88** is installed without the last path component. The resulting directory name will be the root directory of the installed S1C package, unless **mk88** is installed somewhere else. This macro can be used to refer to files belonging to the product, for example a library source file.

Example:

```
DOPRINT = $(PRODDIR)/lib/src/_doprint.c
```

When **mk88** is installed in the directory `/c88/bin` this line expands to:

```
DOPRINT = /c88/lib/src/_doprint.c
```

SHELLCMD

This contains the default list of commands which are local to the SHELL. If a rule is an invocation of one of these commands, a SHELL is automatically spawned to handle it.

TMP_CCPROG

This macro contains the name of the control program. If this macro and the TMP_CCOPT macro are set and the command line argument list for the control program exceeds 127 characters then **mk88** will create a temporary file filled with the command line arguments. **mk88** will call the control program with the temporary file as command input file.

TMP_CCOPT

This macro contains the option for the control program which tells the control program to read a file as command arguments.

Example:

```
TMP_CCPROG = cc88
TMP_CCOPT  = -f
```

$ This macro translates to a dollar sign. Thus you can use "$$" in the makefile to represent a single "$".

There are several dynamically maintained macros that are useful as abbreviations within rules. It is best not to define them explicitly.

$* The basename of the current target.

$< The name of the current dependency file.

$@ The name of the current target.

$? The names of dependents which are younger than the target.

$! The names of all dependents.

The $< and $* macros are normally used for implicit rules. They may be unreliable when used within explicit target command lines. All macros may be suffixed with F to specify the Filename components (e.g. ${*F}, ${@F}). Likewise, the macros $*, $< and $@ may be suffixed by D to specify the directory component.

Functions

A function not only expands but also performs a certain operation. Functions syntactically look like macros but have embedded spaces in the macro name, e.g. '$(match arg1 arg2 arg3)'. All functions are built-in and currently there are five of them: `match`, `separate`, `protect`, `exist` and `nexist`.

The `match` function yields all arguments which match a certain suffix:

```
$(match .obj prog.obj sub.obj mylib.a)
```

will yield

```
prog.obj sub.obj
```

The `separate` function concatenates its arguments using the first argument as the separator. If the first argument is enclosed in double quotes then '\n' is interpreted as a newline character, '\t' is interpreted as a tab, '*ooo*' is interpreted as an octal value (where, *ooo* is one to three octal digits), and spaces are taken literally. For example:

```
$(separate "\n" prog.obj sub.obj)
```

will result in

```
prog.obj
sub.obj
```

Function arguments may be macros or functions themselves. So,

```
$(separate "\n" $(match .obj $!))
```

will yield all object files the current target depends on, separated by a newline string.

The `protect` function adds one level of quoting. This function has one argument which can contain white space. If the argument contains any white space, single quotes, double quotes, or backslashes, it is enclosed in double quotes. In addition, any double quote or backslash is escaped with a backslash.

Example:

```
echo $(protect I'll show you the "protect" function)
```

will yield

```
echo "I'll show you the \"protect\" function"
```

The `exist` function expands to its second argument if the first argument is an existing file or directory.

Example:

```
$(exist test.c cc88 test.c)
```

When the file `test.c` exists it will yield:

```
cc88 test.c
```

When the file `test.c` does not exist nothing is expanded.

The `nexist` function is the opposite of the `exist` function. It expands to its second argument if the first argument is not an existing file or directory.

Example:

```
$(nexist test.src cc88 test.c)
```

Targets

A target entry in the makefile has the following format:

```
target ... : [dependency ...] [; rule]
     [rule]
     ...
```

Any line which does not have leading white space (other than macro definitions) is a 'target' line. Target lines consist of one or more filenames (or macros which expand into same) called targets, followed by a colon (:). The ':' is followed by a list of dependent files. The dependency list may be terminated with a semicolon (;) which may be followed by a rule or shell command.

Special allowance is made on MS-DOS for the colons which are needed to specify files on other drives, so for example, the following will work as intended:

```
c:foo.obj : a:foo.c
```

If a target is named in more than one target line, the dependencies are added to form the target's complete dependency list.

The dependents are the ones from which a target is constructed. They in turn may be targets of other dependents. In general, for a particular target file, each of its dependent files is 'made', to make sure that each is up to date with respect to it's dependents.

The modification time of the target is compared to the modification times of each dependent file. If the target is older, one or more of the dependents have changed, so the target must be constructed. Of course, this checking is done recursively, so that all dependents of dependents of dependents of ... are up-to-date.

To reconstruct a target, **mk88** expands macros and functions, strips off initial white space, and either executes the rules directly, or passes each to a shell or COMMAND.COM for execution.

For target lines, macros and functions are expanded on input. All other lines have expansion delayed until absolutely required (i.e. macros and functions in rules are dynamic).

Special Targets

.DEFAULT:
The rule for this target is used to process a target when there is no other entry for it, and no implicit rule for building it. **mk88** ignores all dependencies for this target.

.DONE:
This target and its dependencies are processed after all other targets are built.

.IGNORE:
Non-zero error codes returned from commands are ignored. Encountering this in a makefile is the same as specifying **-i** on the command line.

.INIT:
This target and its dependencies are processed before any other targets are processed.

.SILENT:
Commands are not echoed before executing them. Encountering this in a makefile is the same as specifying **-s** on the command line.

.SUFFIXES:
The suffixes list for selecting implicit rules. Specifying this target with dependents adds these to the end of the suffixes list. Specifying it with no dependents clears the list.

.PRECIOUS:
Dependency files mentioned for this target are not removed. Normally, **mk88** removes a target file if a command in its construction rule returned an error or when target construction is interrupted.

Rules

A line in a makefile that starts with a TAB or SPACE is a shell line or rule. This line is associated with the most recently preceding dependency line. A sequence of these may be associated with a single dependency line. When a target is out of date with respect to a dependent, the sequence of commands is executed. Shell lines may have any combination of the following characters to the left of the command:

@ will not echo the command line, except if **-n** is used.

- **mk88** will ignore the exit code of the command, i.e. the ERRORLEVEL of MS-DOS. Without this, **mk88** terminates when a non-zero exit code is returned.

+ **mk88** will use a shell or COMMAND.COM to execute the command.

If the '+' is not attached to a shell line, but the command is a DOS command or if redirection is used (<, |, >), the shell line is passed to COMMAND.COM anyway.

mk88 can generate inline temporary files. If a line contains '<<WORD' then all subsequent lines up to a line starting with WORD, are placed in a temporary file. Next, '<<WORD' is replaced by the name of the temporary file.

Example:

```
lk88 -o $@ -f <<EOF
     $(separate "\n" $(match .obj $!))
     $(separate "\n" $(match .a $!))
     $(LKFLAGS)
EOF
```

The three lines between the tags (EOF) are written to a temporary file (e.g. "\tmp\mk2"), and the command line is rewritten as "lk88 -o $@ -f \tmp\mk2".

Implicit Rules

Implicit rules are intimately tied to the .SUFFIXES: special target. Each entry in the .SUFFIXES: list defines an extension to a filename which may be used to build another file. The implicit rules then define how to actually build one file from another. These files are related, in that they must share a common basename, but have different extensions.

If a file that is being made does not have an explicit target line, an implicit rule is looked for. Each entry in the .SUFFIXES: list is combined with the extension of the target, to get the name of an implicit target. If this target exists, it gives the rules used to transform a file with the dependent extension to the target file. Any dependents of the implicit target are ignored.

If a file that is being made has an explicit target, but no rules, a similar search is made for implicit rules. Each entry in the .SUFFIXES: list is combined with the extension of the target, to get the name of an implicit target. If such a target exists, then the list of dependents is searched for a file with the correct extension, and the implicit rules are invoked to create the target.

Examples

This makefile says that `prog.out` depends on two files `prog.obj` and `sub.obj`, and that they in turn depend on their corresponding source files (`prog.c` and `sub.c`) along with the common file `inc.h`.

```
LIB  =       -ls

prog.out:    prog.obj sub.obj
     lk88    prog.obj sub.obj $(LIB) -o prog.out

prog.obj:    prog.c inc.h
     c88     prog.c
     as88    prog.src

sub.obj:     sub.c inc.h
     c88     sub.c
     as88    sub.src
```

The following makefile uses implicit rules (from `mk88.mk`) to perform the same job.

```
LDFLAGS      = -ls
prog.out:    prog.obj sub.obj
prog.obj:    prog.c inc.h
sub.obj:     sub.c inc.h
```

Files

makefile	Description of dependencies and rules.
mk88.mk	Default dependencies and rules.

Diagnostics

mk88 returns an exit status of 1 when it halts as a result of an error. Otherwise it returns an exit status of 0.

6.5 pr88

Name

pr88 IEEE object reader
Displays the contents of a relocatable object file or an absolute file

Synopsis

pr88 [*option*]... *file*
pr88 **-V**
pr88 **-?**

Description

pr88 gives you a high level view of an object file which has been created by a tool from the S1C tool chain. Note that **pr88** is not a disassembler.

Options

Options start with a '-' sign and can be combined after a single '**-**'. There are options to print a specific part of an object file. For example, with option **-h** you can display the header part, the environment part and the AD/extension part as a whole. These parts are small, and you cannot display these parts separately. If you do not specify a part, the default is **-hscegd0i0** (all parts, the debug part and the image part displayed as a table of contents).

Furthermore, there are some additional options by which you can control the output.

Input Control Option

-f *file*

Read command line information from *file*. If *file* is a '**-**', the information is read from standard input.
Use *file* for command line processing. To get around the limits on the size of the command line, it is possible to use command files. These command files contain the options that could not be part of the real command line. Command files can also be generated on the fly, for example by the make utility.
More than one **-f** option is allowed.
Some simple rules apply to the format of the command file:

1. It is possible to have multiple arguments on the same line in the command file.
2. To include whitespace in the argument, surround the argument with either single or double quotes.
3. If single or double quotes are to be used inside a quoted argument, we have to go by the following rules:
   a. If the embedded quotes are only single or double quotes, use the opposite quote around the argument. Thus, if a argument should contain a double quote, surround the argument with single quotes.
   b. If both types of quotes are used, we have to split the argument in such a way that each embedded quote is surrounded by the opposite type of quote.

Example:

```
"This has a single quote ' embedded"
```

or

```
'This has a double quote " embedded'
```

or

```
'This has a double quote " and a single quote '"' embedded"
```

4. Some operating systems impose limits on the length of lines within a text file. To circumvent this limitation it is possible to use continuation lines. These lines end with a backslash and newline. In a quoted argument, continuation lines will be appended without stripping any whitespace on the next line. For non-quoted arguments, all whitespace on the next line will be stripped.

Example:

```
"This is a continuation \
line"
        → "This is a continuation line"

control(file1(mode,type),\
file2(type))
        → control(file1(mode,type),file2(type))
```

5. It is possible to nest command line files up to 25 levels.

Output Control Options

-H or **-?**
Display an explanation of options at `stdout`.

-V
Display version information at `stderr`.

-W*n*
Set output width to *n* columns. Default 128, minimum 78.

-l*n*
Level control, see Section 6.5.3.

-o*file*
Name of the output file, default `stdout`.

-v
Print the selected parts in a verbose form.

-v*n*
Print level *n* verbose, see Section 6.5.3.

-w*n*
Suppress messages above warning level *n*.

Display Options

-c
Print call graphs.

-d
Print all debug info except for the global types.

-d0
Print table of contents for the debug part.

-d*n*
Print debug info from file number *n*.

-e
Print variables with external scope.

-e1
Print variables with external scope and precede symbol name with name of the object file.

-g
Print global types.

-h
Print general file info.

-i
Print all section images.

-i0
Print table of contents for the image part.

-i*n*
Print image of section *n*.

-s
Print section info.

6.5.1 Preparing the Demo Files

There are three files which are used in this chapter to show how you can use **pr88**. These files are:

calc.obj
calc.out
calc.abs

If you want to try the examples yourself, prepare these files by copying the calc example files to a working directory. Be sure that the S1C tools can be found via a search path. Make the files with the following command:

```
cc88 -Ms -nolib startup.asm _copytbl.asm calc.asm -o calc.abs
     s1c88316.dsc -tmp
```

6.5.2 Displaying Parts of an Object File

6.5.2.1 Option -h, display general file info

The **-h** option gives you general information of the file. The invocation:

```
pr88 -h calc.out
```

Gives the following information:

```
File name    = calc.out:
Format       = Relocatable
Produced by  = S1C object linker
Date         = jan 23, 1997 16:35:40h
```

This output speaks for itself. You may combine the **-h** switch with the verbose option:

```
pr88 -hv calc.out
```

The output is extended with more general information of less importance:

```
File name    = calc.out:
Format       = Relocatable
Produced by  = S1C object linker
Date         = jan 23, 1997 16:35:40h
Obj version  = 1.1
Processor    = S1Cs
Address size = 24 bits
Byte order   = Least significant byte at lowest address
Host         = Sun
Part                   File offset    Length
-------------------------------------------------
Header part            0x00000000     0x00000055
AD Extension part      0x00000055     0x00000033
Environment part       0x00000088     0x0000002b
Section part           0x000000b3     0x0000009b
External part          0x0000014e     0x00000098
Debug/type part        0x000001e6     0x000002b8
Data part              0x0000049e     0x000002b8
Module end             0x00000756
```

The table gives you the file offsets and the length of the main object parts.

6.5.2.2 Option -s, display section info

With the **-s** option, you can obtain the section information from an object module. The section **contents** can be obtained with the **-i** option, see Section 6.5.2.7.

```
pr88 -s calc.out
```

```
Section           Size
-------------------------------
.startup_vector   0x000002
.startup          0x000063
.watchdog_vector  0x000002
.watchdog         0x000001
.text             0x00002d
.data             0x000003
.zdata            0x000001
```

Note that the section information is not available any more in a located file. Once located, the separate *sections* are combined to new *clusters*. For an absolute file '**pr88 -s**' will give the *cluster* information:

```
pr88 -s calc.abs
```

```
Section   Size
----------------------
rom       0x0000b9
ram       0x00f800
```

The locate map shows you which section is located in which cluster. Of course, you can also use the verbose option to see all section information available:

```
pr88 -sv calc.out
```

```
Section          Size     Address  Align PageSize Mau Attributes
-------------------------------------------------------------------------------------------
.startup_vector  0x000002 0x000000 0x001 -        -   ReadOnly Execute ZeroPage Space 1 Abs Separate
.startup         0x000063 -        0x001 -        -   ReadOnly Execute ZeroPage Space 1 Cumulate
.watchdog_vector 0x000002 0x000004 0x001 -        -   ReadOnly Execute ZeroPage Space 1 Abs Separate
.watchdog        0x000001 -        0x001 -        -   ReadOnly Execute ZeroPage Space 1 Cumulate
.text            0x00002d -        0x001 -        -   ReadOnly Execute ZeroPage Space 1 Cumulate
.data            0x000003 -        0x001 -        -   Write Space 2 Initialized Cumulate
.zdata           0x000001 -        0x001 -        -   Write Space 2 Cleared Cumulate
```

The first two columns give you the section name and the section size. The column 'Address' gives you the section address, or a '-' if the section is still relocatable. The section alignment is always 1 for the S1C. The page size is valid only for the short sections. MAU is the minimum addressable unit of an address space (in bits). There are two main groups of section attributes, the allocation attributes, used by the locator and the overlap attributes, used by the linker:

Allocation attributes

Write	Must be located in ram
ReadOnly Execute Space *num* Abs	May be located in rom May be located in rom Must be located in addressing mode *num* Already located by the assembler
Cleared	Section must be initialized to '0'
Initialized Scratch	Section must be copied from ram to rom Section is not filled or cleared

Overlap attributes

MaxSize Unique	Use largest length encountered Only one section with this name allowed
Cumulate	Concatenate sections with the same name to one bigger section
Overlay	Sections with the name *name@func* must be combined to one section *name*, according to the rules for *func* obtained from the call graph.
Separate	Sections are not linked

6.5.2.3 Option -c, display call graphs

The call graph is used by the linker overlaying algorithm. Once a file is linked and overlaying is done, the call graph information is removed from the object file. If you try to see the call graph in `calc.out` you will get the message 'No call graph found'.

The file `calc.obj` is not yet linked. You can use this file to see what a call graph looks like:

```
pr88 -c calc.obj
```

Because the `calc` example does not contain any sections which need to be overlaid you will again get the message 'No call graph found'. The following is just an example of what a call graph could look like:

```
Call graph(s)
=============

Call graph 0:

main()
  ->See call graph 1
  ->See call graph 4
  ->See call graph 2
  _exit()
  print_str()
  clear_screen()

Call graph 1:

queens?find_legal_row()
  ->See call graph 1
  ->See call graph 2
  abs()
  ->See call graph 3
```

Each call graph consists of a function (`main` in graph 0), followed by a list of functions and/or other graphs, which are called by the first function. The functions and call graphs called by this function are indented by two spaces. If a function calls other functions, those functions are listed again with another indentation of two spaces.

As you can see, there are references from one call graph to another. Call graph 1 even calls itself!! This means that function `find_legal_row()` is a recursive function. If you use the verbose switch the output is somewhat nicer:

```
main()
  |
  +--->See call graph 1
  |
  +--->See call graph 4
  |
  +--->See call graph 2
  |
  +--exit()
  |
  +--print_str()
  |
  +--clear_screen()
```

The function `find_legal_row` from call graph 1 is a static function. In order to avoid name conflicts, the source name is added to this function name.

If you want a call graph with resolved call graph references, you can use the linker to generate one:

```
lk88 -o call.out -Mcr calc.obj
```

Option **-M** tells the linker to generate a `.lnl` file. This file contains the call graph in the verbose layout. Option **-c** causes the linker to generate a `.cal` file. This file contains also the (same) call graph, but in the compact (non verbose) layout. Option **-r** tells the linker that this is an incremental link.

6.5.2.4 Option -e, display external part

In the external part of an object file, you can find all symbols used at link time. These symbols have an external scope. With the **-e** option (or **-e0**) **pr88** displays the external symbols:

```
pr88 -e calc.out
```

```
Variable     S  Address/Size
-----------------------------
__start_cpt  I  .startup + 0x00
__START      I  .startup + 0x00
__exit       I  .startup + 0x20
__copytable  I  .startup + 0x22
_main        I  .text + 0x20
__lc_es      X  -
__lc_cp      X  -
```

With option **-e1** also the name of the output object file is displayed.

```
pr88 -e1 calc.out
```

```
Variable              S  Address/Size
-------------------------------------
calc.out:__start_cpt  I  .startup + 0x00
calc.out:__START      I  .startup + 0x00
calc.out:__exit       I  .startup + 0x20
calc.out:__copytable  I  .startup + 0x22
calc.out:_main        I  .text + 0x20
calc.out:__lc_es      X  -
calc.out:__lc_cp      X  -
```

The first column contains the name of the symbol. In general, this symbol is a high level symbol with an 'F' added at the front. The next column gives you the symbol status. This can be **I** for a defined symbol, and **X** for a symbol which is referred to, but which is not yet defined. In the last column you can find the symbols address. If this address is still relocatable, the section offsets are printed in the form '*section* + offset'. If a symbol has already received an absolute address, this address is printed. Symbols that are not yet defined (marked with a **X**) have a dash printed as address, indicating *unknown*.

You can add the verbose option as usual. With verbose on more information is printed:

```
pr88 -ev calc.out
```

```
Variable       S       Type    Attrib  MAU     Amod    Address/Size
-------------------------------------------------------------------------
__start_cpt    I       -       -       8       1       .startup + 0x00
__START        I       -       -       8       1       .startup + 0x00
__exit         I       -       -       8       1       .startup + 0x20
__copytable    I       -       -       8       1       .startup + 0x22
_main          I       -       -       8       1       .text + 0x20
__lc_es        X       -       -       8       2       -
__lc_cp        X       -       -       8       2       -
```

Four additional columns appear. The Type column gives you the symbol type, if available. You can find the meaning of the types in the global type part, Section 6.5.2.5. The global types are used to type check the symbols during linking. The Attribute column specifies the attribute of the symbol, if available. For example, the attribute value 0x0020 indicates that the symbol is generated by the assembler. The MAU column indicates the minimum addressable unit in bits. So, MAU 8 means the symbol is 8-bit addressable. The Amod column lists the addressing mode of the symbol.

6.5.2.5 Option -g, display global type information

The linker uses the global type information to check on type mismatches of the symbols in the external part. This information is always available, unless you explicitly suppress the generation of these types with option **-gn** at compile time. Of course, type checking can only be done if the types are available. The global types in `calc.out`:

```
pr88 -g calc.out
```

In this example you will get the message 'No global types available'. The following is just an example of what the global type information could look like:

```
Tp#     Mnem    Name    Entry
---------------------------------
101     X       -       0, T10, 0, 0
102     X       -       0, T1, 0, 0
103     X       -       0, T1, 0, 1, T104
104     P       -       T105
105     n       -       T2, 1
106     X       -       0, T1, 0, 1, T10
107     X       -       0, T10, 0, 1, T10
108     X       -       0, T1, 0, 2, T109, T109
109     T       Byte    T3
10a     X       -       0, T1, 0, 1, T109
...
10f     X       -       0, T1, 0, 3, T12, T110, T12
110     O       -       T111
111     n       -       T2, 0
112     Z       -       T2, 13
113     Z       -       T2, 7
```

In the first column you find the type index. This is the number by which the type is referred to. This number is always a hexadecimal number. Numbering starts at 0x101, because the indices less than 0x100 are reserved for, so-called, 'basic types'. The second column contains the type mnemonic. This mnemonic defines the new 'high level' type. In the Name column you will find the name for the type, if any.

The last column contains type parameters. They tell you which (basic) types a high level type is based on and give other parameters such as modes and sizes. Types are preceded by a **T**. So, in the example above, type 105 is based upon type 2 (T2 in the parameter list) and type 103 is based upon type 1 and type 104.

In the next table you can find an overview of the basic types:

Type index	Type	Meaning
1	void	-
2	char	8 bits signed
3	unsigned char	8 bits unsigned
4	short	16 bits signed
5	unsigned short	16 bits unsigned
6	long	32 bits signed
7	unsigned long	32 bits unsigned
10	float	32 bit floating point
11	double	64 bit floating point
16	int	16 bits signed
17	unsigned int	16 bits unsigned

The type mnemonics define the class of the newly created type. The next table shows the type mnemonics with a short description:

Mnemonic	Description	Parameters
G	generalized structure	size, [member, T*index*, offset, size]...
N	enumerated type	[name, value]...
n	pointer qualifier	T*index*, memspace
O	small pointer	T*index*
P	large pointer	T*index*
Q	type qualifier	q-bits, T*index*
S	structure	size, [member, T*index*, offset]...
T	typedef	T*index*
t	compiler generated type	T*index*
U	union	size, [member, T*index*, offset]...
X	function	x-bits, T*index*, 0, nbr-arg, [T*index*]...
Z	array	T*index*, upper-bound
g	bit type	sign, nbr-of-bits

The T*index* for mnemonic n, O, P, Q, T, t and Z are the types upon which the new type is built. The T*index* for the union and the structures are the type indices for the members. For the function type, the first T*index* is the return type of the function. The second T*index* is repeated for each parameter, and gives the type of each parameter. The value -1 (0xffffffff) always means 'unknown'. This can occur with a function type if the number of parameters is unknown, or with an array if the upper bound in unknown. The sizes and offset for the generalized structure are in bits. The first size is the size of the structure, the second size is the size for the member.

The type information obtained with the **-g** switch has no verbose equivalent.

6.5.2.6 Option -d, display debug information

The **-d** switch has two variants. With **-d0** you get a table of contents:

```
pr88 -d0 calc.out

Choose option -d with the number of the file:
 1 - startup
 2 - _copytbl
 3 - calc
```

Now, you can use **-d***n* to examine a single (linked) file. For instance, **-d3** shows you only the debug info of `calc.obj`. It is also possible to see all debug info, by using option **-d** without a value.

The **-d** switch without the verbose option **-v** shows you only local variables and procedure information. If you combine the **-d** switch with the verbose switch **-v**, also local type info, line numbers, stack update information and more procedure information is displayed.

In the example you are using the verbose switch. Where required, the remark 'Only with verbose on' will be given.

```
pr88 -d3v calc.out
```

The object reader starts with a header, followed by the local type information:

```
                        ****************************
                        *   O b j e c t   c a l c  *
                        ****************************

                            M o d u l e   i n f o
                            =====================

Type info calc:
===============

No local types available
```

This type info is only printed if you use the verbose option **-v**. The information found in this table is exactly the same as the information explained for the global type information, see Section 6.5.2.5.

After the local types, you will find the local symbols.

```
Symbols calc:
=============

Variable        S       Type    Attrib  MAU     Amod    Address/Size
-----------------------------------------------------------------------
_MODEL          N       -       0x0010  0       0       -
_MODEL          N       -       0x0010  0       0       -
_factorial      N       -       0x0020  8       1       -
_compute        N       -       0x0020  8       1       -
_val            N       -       0x0020  8       2       -
_zero           N       -       0x0020  8       2       -
_cll            N       -       0x0020  8       2       -
```

The value for the symbol status in the external part was an **I** or an **X**. Here, you can see a new letter. The **N** stands for a local symbol. Other possible entries can have the letter **G** or **S**. They are no symbols, but procedures. These procedures are printed at this place in order to define their relative position. The actual procedure information is given in the next block of information. Here you can find the additional procedure information. The procedure block is printed only if you use the verbose switch:

```
Procedures calc:
================

No procedures
```

The following is an example of some procedures:

```
Name            S  Additional information
------------------------------------------
main            G  0x00, 0x00, T101, QUEENS_PR + 0x00,
                   ( QUEENS_PR + 0x49 ) - 0x01
find_legal_row  S  0x00, 0x00, T120, QUEENS_PR + 0x49,
                   ( QUEENS_PR + 0x156 ) - 0x01
display_board   S  0x00, 0x00, T10a, QUEENS_PR + 0x156,
                   ( QUEENS_PR + 0x2a4 ) - 0x01
display_field   S  0x00, 0x00, T121, QUEENS_PR + 0x2a4,
                   ( QUEENS_PR + 0x302 ) - 0x01
display_status  S  0x00, 0x00, T103, QUEENS_PR + 0x302,
                   ( QUEENS_PR + 0x31d ) - 0x01
```

The first two columns are the same as those in the local variable table. The **G** stands for an external (global) function, the **S** for a static (local) function.

Each function has 5 parameters with the following meaning:

- param #1 Frame type, not used
- param #2 Frame size, the distance from the stack pointer before the function call to the stack position just after the local variables.
- param #3 The type of the function
- param #4 The start address of the function. In a relocatable object the syntax '*section* + offset' is used.
- param #5 The last function address. See also param #4.

Next in the debug info is the line number information and the stack information. Both items are only printed if you had turned the verbose switch on:

```
Lines include/stdarg.h:
=======================
No line info available
Lines include/stdio.h:
======================
No line info available
```

```
Lines queens.c:
===============

Address              | Line  Address              | Line  Address     ...
-------------------------  -------------------------  -----------------
QUEENS_PR + 0x000000 | 52    QUEENS_PR + 0x0000c2 | 90    QUEENS_PR + ...
QUEENS_PR + 0x000000 | 53    QUEENS_PR + 0x0000d9 | 101   QUEENS_PR + ...
QUEENS_PR + 0x000006 | 55    QUEENS_PR + 0x0000d9 | 103   QUEENS_PR + ...
        .                          .                          .
        .                          .                          .
        .                          .                          .
QUEENS_PR + 0x0000bd | 98    QUEENS_PR + 0x00018e | 133   QUEENS_PR + ...
QUEENS_PR + 0x0000c0 | 99    QUEENS_PR + 0x000190 | 136   QUEENS_PR + ...
QUEENS_PR + 0x0000c2 | 100   QUEENS_PR + 0x00019f | 137

Stack info include/stdarg.h:
============================
No stack info available

Stack info include/stdio.h:
===========================
No stack info available

Stack info queens.c:
====================
No stack info available
```

The stack info gives the actual stack position for each executable address. This value is measured from the start position, just after the functions local variables to the actual stack position. If you push one byte on stack, the delta will be increased by one.

The debug info per module ends with a block for each function. Within this block the local variables per function are displayed:

```
                        P r o c e d u r e   i n f o
                        ===========================

Procedure find_legal_row:
=========================

Symbols find_legal_row:
=======================

Variable  S  Type    Attrib  Mau  Amod  Address/Size
-----------------------------------------------------
accepted  N  0x0109  0x0004   0     0   QUEENS_DA + 0x09
row       N  0x0109  0x0805   0     0   0x02
col       N  0x0109  0x0805   0     0   0x03
chk_row   N  0x0109  0x0005   0     0   0x01
chk_col   N  0x0109  0x0005   0     0   0x00

                E n d   o f   p r o c e d u r e   i n f o
                =========================================
```

6.5.2.7 Option -i, display the section images

As with the **-d** option, you can ask a table with available section images by specifying option **-i0**:

```
pr88 -i0 calc.out

Choose option -i with the number of the section:
 1 - .startup_vector
 2 - .startup
 3 - .watchdog_vector
 4 - .watchdog
 5 - .text
 6 - .data
 7 - .zdata
```

You can select the image to display by specifying the image number:

```
pr88 -i5 calc.out

Section .text:
==============

02 32 05 e3 ce 00 01 c4 f8 b0 cf 88 f3 f0 50 b4
cf d8 ce a1 51 d8 ce e1 cf 00 b1 cc a9 01 cf f8
rr rr rr rr rr rr rr rr rr rr rr rr rr
```

It is also possible to get the section offsets or absolute addresses by specifying the verbose flag:

```
pr88 -i5v calc.out

Section .text:
==============

000000  02 32 05 e3 ce 00 01 c4 f8 b0 cf 88 f3 f0 50 b4 .2............P.
000010  cf d8 ce a1 51 d8 ce e1 cf 00 b1 cc a9 01 cf f8 ....Q...........
000020  rr rr rr rr rr rr rr rr rr rr rr rr rr           .............
```

The dump always shows the hexadecimal byte value per address. Sometimes however, this is not possible. First of all, it is possible that a certain byte cannot be determined because it is not yet relocated. In this case the byte is represented as **rr**.

Secondly, it is possible that there is no section image allowed. This is for instance the case for sections that are cleared during startup. After the invocation (verbose on) the reader prints:

```
pr88 -i7v calc.out

Section .zdata:
===============

No image allowed, cleared during startup
```

It is possible that you read an absolute file. In the absolute file it is possible to combine different sections to new clusters. These clusters do not have the same attributes as the sections and the reader does no longer know where the overlay area is positioned:

```
pr88 -v -i1 calc.abs

Section rom:
============

000000  00 53 f9 ss 00 02 02 00 f0 00 00 00 00 00 00 00 .S..............
000010  01 01 00 f0 01 00 00 b6 00 00 00 03 00 ss ss ss .......2........
000020  ss ss ss ss ss ss 02 32 05 e3 ce 00 01 c4 f8 b0 ....P.....Q.....
000030  cf 88 f3 f0 50 b4 cf d8 ce a1 51 d8 ce e1 cf 00 ................
....
```

As you see, the reader only prints bytes that it actually can read from the object file. The **ss** in the dump means *scratch* memory. It may or may not be initialized by the start-up code. This information is not available anymore to the reader. The start-up code can use a locator generated table to get the information. See Chapter 4, "Locator".

6.5.3 Viewing an Object at Lower Level

6.5.3.1 Object Layers

As with the well known OSI layer model for communication, you can also distinguish layers in an object file. The object file is a medium for the compiler which lets the compiler communicate with the debugger or the target board. The lowest level can be classified as mass storage, mostly the disc. The lowest viewable level for the readers concern are the raw bytes.

pr88 knows this layer as *level 0*.

Of course, the bytes in level 0 have a meaning. Because the object format is an format according to IEEE 695, the object file is a collection of MUFOM commands. The general idea is, that an object producing tool sends commands to a object consuming tool. These commands are described in detail by the official IEEE standard[1]. The raw bytes from level 0 appear to be encoded MUFOM commands. The MUFOM commands are interpreted in a layer just above the raw bytes layer.

pr88 knows this layer as *level 1*.

The next layer is the MUFOM environment, the type and section tables are built, values are assigned, attributes are set just by performing the MUFOM commands. The IEEE document describes also some predefined meanings about scope, section attributes naming conventions for MUFOM variables. This knowledge is available in the highest MUFOM layer.

pr88 knows this layer as *level 2*.

With these first layers, the compiler and debugger/target board have a perfect communication channel. The next layers (not supported by the reader at this moment) define a protocol between compiler and debugger about target and language specific information.

In the next sections you can find some examples about the use of the reader at lower levels. Until now, you used the default level of the reader, level 2.

6.5.3.2 The Level Option -ln

Level 1

Switching to another level is simple. You can use the **-l** option with the level you want to see. As an example, the section part of `calc.out` at level 1:

```
pr88 -l1 -s calc.out
```

```
ST:     1, RXAZS, .startup_vector
AS:     L1, 0x0
AS:     S1, 0x2
ST:     2, RXZC, .startup
AS:     S2, 0x63
ST:     3, RXAZS, .watchdog_vector
AS:     L3, 0x4
AS:     S3, 0x2
ST:     4, RXZC, .watchdog
AS:     S4, 0x1
ST:     5, RXZC, .text
AS:     S5, 0x2d
ST:     6, WIY2C, .data
AS:     S6, 0x3
ST:     7, WBY2C, .zdata
AS:     S7, 0x1
```

[1] IEEE Trial Use Standard for Microprocessor Universal Format for Object Modules (IEEE std. 695), IEEE Technical Committee on Microcomputers and Microprocessors of the IEEE Computer Society, 1990.

If you are not familiar with the MUFOM commands, you can use the verbose switch. The abbreviated commands such as AS, SA or ST are expanded to *Assignment*, *Section alignment* and *Section type*:

```
pr88 -v -l1 -s calc.out
```

```
ST:     Section type:
        Nbr = 1, type = RXAZS, name = .startup_vector
AS:     Assignment:
        Variable = L1, expression = 0x0
AS:     Assignment:
        Variable = S1, expression = 0x2
.
.
ST:     Section type:
        Nbr = 7, type = WBY2C, name = .zdata
AS:     Assignment:
        Variable = S7, expression = 0x1
```

The L*n* and S*n* MUFOM variables are defined as the address and the size of section *n*. At level 2 you saw (refer to Section 6.5.2.2) that the level 2 view did not mention the L and S variables, because at level 2 the meaning of the L and S variables are known!

Level 0

Switching to level 0 is accomplished by using **-l0** (as you expected):

```
pr88 -l0s calc.out
```

```
e6 01 d2 d8 c1 da d3 0f 2e 73 74 61 72 74 75 70 5f
76 65 63 74 6f 72
e2 cc 01 81 00
e2 d3 01 02
...
e6 07 d7 c2 d9 02 c3 06 2e 7a 64 61 74 61
e2 d3 07 01
```

The bytes are printed in the MUFOM command structure. It should be easy to find the encoding for the used MUFOM commands. You can use the verbose switch if you want to see file offsets:

```
pr88 -l0vs calc.out
```

```
0000b3   e6 01 d2 d8 c1 da d3 0f 2e 73 74 61 72 74 75 70 5f .........startup_
         76 65 63 74 6f 72                                  vector
0000ca   e2 cc 01 81 00                                     .....
0000cf   e2 d3 01 02                                        ....
....
00013c   e6 07 d7 c2 d9 02 c3 06 2e 7a 64 61 74 61         .........zdata
00014a   e2 d3 07 01                                        ....
```

Viewing Mixed Levels

You can also mix the levels. It is for instance possible to see level 0 and 1 together by specifying option **-l01** (equivalent to **-l10** or **-l0 -l1**):

```
pr88 -sl01 calc.out
```

```
ST:     1, RXAZS, .startup_vector
        e6 01 d2 d8 c1 da d3 0f 2e 73 74 61 72 74 75 70 5f
        76 65 63 74 6f 72
AS:     L1, 0x0
        e2 cc 01 81 00
AS:     S1, 0x2
        e2 d3 01 02
.
.
.
ST:     7, WBY2C, .zdata
        e6 07 d7 c2 d9 02 c3 06 2e 7a 64 61 74 61
AS:     S7, 0x1
        e2 d3 07 01
```

And of course, you can turn on the verbose switch. The switch between level 0 and level 1 is done per MUFOM command. This is because a MUFOM command is the smallest unit at level 1.

If you should display level 1 and 2, the switch is made per object part, because the object parts are the smallest units at level 2. It is not possible to show the results of all section related commands before all these commands are executed:

```
pr88 -s -l1 -l2 calc.out
```

```
ST:     1, RXAZS, .startup_vector
AS:     L1, 0x0
AS:     S1, 0x2
.
.
.
ST:     7, WBY2C, .zdata
AS:     S7, 0x1
Section           Size
------------------------------
.startup_vector   0x000002
.startup          0x000063
.watchdog_vector  0x000002
.watchdog         0x000001
.text             0x00002d
.data             0x000003
.zdata            0x000001
```

6.5.3.3 The Verbose Option -vn

As you have read in Section 6.5.3.2, you can switch to a lower level with the level switch **-l***n*. If you want a verbose printout, you can use the **-v** option.

It is also possible to specify **-v0** to see a verbose output of level 0, option **-v***n* is a shorthand for options **-v -l***n* (or **-vl***n*). The new notation has the advantage that if you want a mixed level output, you are able to choose the verbose option **per level**. You may specify **-l0 -v1**, and you get a non verbose level 0 and a verbose level 1:

```
pr88 -sl0v1 calc.out
ST:     Section type:
        Nbr = 1, type = RXAZS, name = .startup_vector
        e6 01 d2 d8 c1 da d3 0f 2e 73 74 61 72 74 75 70 5f
        76 65 63 74 6f 72
AS:     Assignment:
        Variable = L1, expression = 0x0
        e2 cc 01 81 00
AS:     Assignment:
        Variable = S1, expression = 0x2
        e2 d3 01 02
.
.
.
ST:     Section type:
        Nbr = 7, type = WBY2C, name = .zdata
        e6 07 d7 c2 d9 02 c3 06 2e 7a 64 61 74 61
AS:     Assignment:
        Variable = S7, expression = 0x1
        e2 d3 07 01
```

The general verbose switch **-v** (without a number) makes all selected levels verbose. The verbose switch **-v***n* selects level *n* and makes only level *n* verbose.

APPENDIX A C COMPILER ERROR MESSAGES

Errors start with an error type, followed by a number and a message. The error type is indicated by a letter:

- I information
- E error
- F fatal error
- S internal compiler error
- W warning

Frontend

F 1: evaluation expired
Your product evaluation period has expired.

W 2: unrecognized option: '*option*'
The option you specified does not exist. Check the invocation syntax for the correct option.

E 4: expected *number* more '#endif'
The preprocessor part of the compiler found the '#if', '#ifdef' or '#ifndef' directive but did not find a corresponding '#endif' in the same source file. Check your source file that each '#if', '#ifdef' or '#ifndef' has a corresponding '#endif'.

E 5: no source modules
You must specify at least one source file to compile.

F 6: cannot create "*file*"
The output file or temporary file could not be created. Check if you have sufficient disk space and if you have write permissions in the specified directory.

F 7: cannot open "*file*"
Check if the file you specified really exists. Maybe you misspelled the name, or the file is in another directory.

F 8: attempt to overwrite input file "*file*"
The output file must have a different name than the input file.

E 9: unterminated constant character or string
This error can occur when you specify a string without a closing double-quote (") or when you specify a character constant without a closing single-quote ('). This error message is often preceded by one or more E 19 error messages.

F 11: file stack overflow
This error occurs if the maximum nesting depth (50) of file inclusion is reached. Check for #include files that contain other #include files. Try to split the nested files into simpler files.

F 12: memory allocation error
All free space has been used. Free up some memory by removing any resident programs, divide the file into several smaller source files, break expressions into smaller subexpressions or put in more memory.

W 13: prototype after forward call or old style declaration - ignored
Check that a prototype for each function is present before the actual call.

E 14: ';' inserted
An expression statement needs a semicolon. For example, after `++i` in `{ int i; ++i }`.

E 15: missing filename after -o option
The **-o** option must be followed by an output filename.

E 16: bad numerical constant
A constant must conform to its syntax. For example, `08` violates the octal digit syntax. Also, a constant may not be too large to be represented in the type to which it was assigned. For example, `int i = 0x1234567890;` is too large to fit in an integer.

E 17: string too long
This error occurs if the maximum string size (1500) is reached. Reduce the size of the string.

E 18: illegal character (0x*hexnumber*)
The character with the hexadecimal ASCII value 0x*hexnumber* is not allowed here. For example, the '#' character, with hexadecimal value 0x23, to be used as a preprocessor command, may not be preceded by non-white space characters. The following is an example of this error:

```
char *s = #S ;  // error
```

E 19: newline character in constant
The newline character can appear in a character constant or string constant only when it is preceded by a backslash (\). To break a string that is on two lines in the source file, do one of the following:

- End the first line with the line-continuation character, a backslash (\).
- Close the string on the first line with a double quotation mark, and open the string on the next line with another quotation mark.

E 20: empty character constant
A character constant must contain exactly one character. Empty character constants ('') are not allowed.

E 21: character constant overflow
A character constant must contain exactly one character. Note that an escape sequence (for example, **\t** for tab) is converted to a single character.

E 22: '#define' without valid identifier
You have to supply an identifier after a '#define'.

E 23: '#else' without '#if'
'#else' can only be used within a corresponding '#if', '#ifdef' or '#ifndef' construct. Make sure that there is a '#if', '#ifdef' or '#ifndef' statement in effect before this statement.

E 24: '#endif' without matching '#if'
'#endif' appeared without a matching '#if', '#ifdef' or '#ifndef' preprocessor directive. Make sure that there is a matching '#endif' for each '#if', '#ifdef' and '#ifndef' statement.

E 25: missing or zero line number
'#line' requires a non-zero line number specification.

E 26: undefined control
A control line (line with a '*#identifier*') must contain one of the known preprocessor directives.

W 27: unexpected text after control
'#ifdef' and '#ifndef' require only one identifier. Also, '#else' and '#endif' only have a newline. '#undef' requires exactly one identifier.

W 28: empty program
The source file must contain at least one external definition. A source file with nothing but comments is considered an empty program.

E 29: bad '#include' syntax
A '#include' must be followed by a valid header name syntax. For example, `#include <stdio.h` misses the closing '>'.

E 30: include file "*file*" not found
Be sure you have specified an existing include file after a '#include' directive. Make sure you have specified the correct path for the file.

E 31: end-of-file encountered inside comment
The compiler found the end of a file while scanning a comment. Probably a comment was not terminated. Do not forget a closing comment '*/' when using ANSI-C style comments.

E 32: argument mismatch for macro "*name*"
The number of arguments in invocation of a function-like macro must agree with the number of parameters in the definition. Also, invocation of a function-like macro requires a terminating ")" token. The following are examples of this error:

```
#define A(a) 1
int i = A(1,2); /* error */
#define B(b) 1
int j = B(1;    /* error */
```

E 33: "*name*" redefined
The given identifier was defined more than once, or a subsequent declaration differed from a previous one. The following examples generate this error:

```
int  i;
char i;         /* error */
main()
{
}
main()
{
        int j;
        int j;  /* error */
}
```

W 34: illegal redefinition of macro "*name*"
A macro can be redefined only if the body of the redefined macro is exactly the same as the body of the originally defined macro.
This warning can be caused by defining a macro on the command line and in the source with a '#define' directive. It also can be caused by macros imported from include files. To eliminate the warning, either remove one of the definitions or use an '#undef' directive before the second definition.

E 35: bad filename in '#line'
The string literal of a #line (if present) may not be a "wide-char" string. So, `#line 9999 L"t45.c"` is not allowed.

W 36: 'debug' facility not installed
'#pragma debug' is only allowed in the debug version of the compiler.

W 37: attempt to divide by zero
A divide or modulo by zero was found. Adjust the expression or test if the second operand of a divide or modulo is zero.

E 38: non integral switch expression
A `switch` condition expression must evaluate to an integral value. So, `char *p = 0; switch (p)` is not allowed.

F 39: unknown error number: *number*
This error may not occur. If it does, contact your local Seiko Epson office and provide them with the exact error message.

W 40: non-standard escape sequence
Check the spelling of your escape sequence (a backslash, `\`, followed by a number or letter), it contains an illegal escape character. For example, `\c` causes this warning.

E 41: '#elif' without '#if'
The '#elif' directive did not appear within an '#if', '#ifdef or '#ifndef' construct. Make sure that there is a corresponding '#if', '#ifdef' or '#ifndef' statement in effect before this statement.

E 42: syntax error, expecting parameter type/declaration/statement
A syntax error occurred in a parameter list a declaration or a statement. This can have many causes, such as, errors in syntax of numbers, usage of reserved words, operator errors, missing parameter types, missing tokens.

E 43: unrecoverable syntax error, skipping to end of file
The compiler found an error from which it could not recover. This error is in most cases preceded by another error. Usually, error E 42.

I 44: in initializer "*name*"
Informational message when checking for a proper constant initializer.

E 46: cannot hold that many operands
The value stack may not exceed 20 operands.

E 47: missing operator
An operator was expected in the expression.

E 48: missing right parenthesis
')' was expected.

W 49: attempt to divide by zero - potential run-time error
An expression with a divide or modulo by zero was found. Adjust the expression or test if the second operand of a divide or modulo is zero.

E 50: missing left parenthesis
'(' was expected.

E 51: cannot hold that many operators
The state stack may not exceed 20 operators.

E 52: missing operand
An operand was expected.

E 53: missing identifier after 'defined' operator
An identifier is required in a `#if defined`(*identifier*).

E 54: non scalar controlling expression
Iteration conditions and 'if' conditions must have a scalar type (not a `struct`, `union` or a pointer). For example, after `static struct {int i;} si = {0};` it is not allowed to specify `while (si) ++si.i;`.

E 55: operand has not integer type
The operand of a '#if' directive must evaluate to an integral constant. So, `#if 1`. is not allowed.

W 56: '<*debugoption*><*level*>' no associated action
This warning can only appear in the debug version of the compiler. There is no associated debug action with the specified debug option and level.

W 58: invalid warning number: *number*
The warning number you supplied to the **-w** option does not exist. Replace it with the correct number.

F 59: sorry, more than number errors
Compilation stops if there are more than 40 errors.

E 60: label "*label*" multiple defined
A label can be defined only once in the same function. The following is an example of this error:

```
f()
{
lab1:
lab1:           /* error */
}
```

E 61: type clash
The compiler found conflicting types. For example, a `long` is only allowed on `int` or `double`, no specifiers are allowed with `struct`, `union` or `enum`. The following is an example of this error:

```
unsigned signed int i; /* error */
```

E 62: bad storage class for "*name*"
The storage class specifiers `auto` and `register` may not appear in declaration specifiers of external definitions. Also, the only storage class specifier allowed in a parameter declaration is `register`.

E 63: "*name*" redeclared
The specified identifier was already declared. The compiler uses the second declaration. The following is an example of this error:

```
struct T { int i; };
struct T { long j; };  /* error */
```

E 64: incompatible redeclaration of "*name*"
The specified identifier was already declared. All declarations in the same function or module that refer to the same object or function must specify compatible types. The following is an example of this error:

```
f()
{
        int i;
        char i; /* error */
}
```

W 66: function "*name*": variable "*name*" not used
A variable is declared which is never used. You can remove this unused variable or you can use the **-w66** option to suppress this warning.

W 67: illegal suboption: *option*
The suboption is not valid for this option. Check the invocation syntax for a list of all available suboptions.

W 68: function "*name*": parameter "*name*" not used
A function parameter is declared which is never used. You can remove this unused parameter or you can use the **-w68** option to suppress this warning.

E 69: declaration contains more than one basic type specifier
Type specifiers may not be repeated. The following is an example of this error:

```
int char i;     /* error */
```

E 70: 'break' outside loop or switch
A `break` statement may only appear in a `switch` or a loop (`do`, `for` or `while`). So, `if (0) break;` is not allowed.

E 71: illegal type specified
The type you specified is not allowed in this context. For example, you cannot use the type `void` to declare a variable. The following is an example of this error:

```
void i; /* error */
```

W 72: duplicate type modifier
Type qualifiers may not be repeated in a specifier list or qualifier list. The following is an example of this warning:

```
{ long long i; }        /* error */
```

E 73: object cannot be bound to multiple memories
Use only one memory attribute per object. For example, specifying both rom and ram to the same object is not allowed.

E 74: declaration contains more than one class specifier
A declaration may contain at most one storage class specifier. So, `register auto i;` is not allowed.

E 75: 'continue' outside a loop
`continue` may only appear in a loop body (`do`, `for` or `while`). So, `switch (i) {default: continue;}` is not allowed.

E 76: duplicate macro parameter "*name*"
The given identifier was used more than one in the formatl parameter list of a macro definition. Each macro parameter must be uniquely declared.

E 77: parameter list should be empty
An identifier list, not part of a function definition, must be empty.
For example, `int f ( i, j, k );` is not allowed on declaration level.

E 78: 'void' should be the only parameter
Within a function prototype of a function that does not except any arguments, `void` may be the only parameter. So, `int f(void, int);` is not allowed.

E 79: constant expression expected
A constant expression may not contain a comma. Also, the bit field width, an expression that defines an `enum`, array-bound constants and `switch` case expressions must all be integral constant expressions.

E 80: '#' operator shall be followed by macro parameter
The '#' operator must be followed by a macro argument.

E 81: '##' operator shall not occur at beginning or end of a macro
The '##' (token concatenation) operator is used to paste together adjacent preprocessor tokens, so it cannot be used at the beginning or end of a macro body.

W 86: escape character truncated to 8 bit value
The value of a hexadecimal escape sequence (a backslash, `\`, followed by a 'x' and a number) must fit in 8 bits storage. The number of bits per character may not be greater than 8. The following is an example of this warning:

```
char c = '\xabc';       /* error */
```

E 87: concatenated string too long
The resulting string was longer than the limit of 1500 characters.

W 88: "*name*" redeclared with different linkage
The specified identifier was already declared. This warning is issued when you try to redeclare an object with a different basic storage class, and both objects are not declared extern or static. The following is an example of this warning:

```
int i;
int i();        /* error E 64 and warning */
```

E 89: illegal bitfield declarator
A bit field may only be declared as an integer, not as a pointer or a function for example. So, `struct {int *a:1;} s;` is not allowed.

E 90: #error *message*
The *message* is the descriptive text supplied in a '#error' preprocessor directive.

W 91: no prototype for function "*name*"
Each function should have a valid function prototype.

W 92: no prototype for indirect function call
Each function should have a valid function prototype.

I 94: hiding earlier one
Additional message which is preceded by error E 63. The second declaration will be used.

F 95: protection error: *message*
Something went wrong with the protection key initialization. The message could be: "Key is not present or printer is not correct.", "Can't read key.", "Can't initialize key.", or "Can't set key-model".

E 96: syntax error in #define
`#define id(` requires a right-parenthesis ')'.

E 97: "..." incompatible with old-style prototype
If one function has a parameter type list and another function, with the same name, is an old-style declaration, the parameter list may not have ellipsis. The following is an example of this error:

```
int f(int, ...);
int f();                /* error, old-style */
```

E 98: function type cannot be inherited from a typedef
A `typedef` cannot be used for a function definition. The following is an example of this error:

```
typedef int INTFN();
INTFN f {return (0);}  /* error */
```

F 99: conditional directives nested too deep
'#if', '#ifdef' or '#ifndef' directives may not be nested deeper than 50 levels.

E 100: case or default label not inside switch
The `case:` or `default:` label may only appear inside a `switch`.

E 101: vacuous declaration
Something is missing in the declaration. The declaration could be empty or an incomplete statement was found. You must declare array declarators and `struct`, `union`, or `enum` members. The following are examples of this error:

```
int ;                   /* error */
static int a[2] = { };  /* error */
```

E 102: duplicate case or default label
Switch `case` values must be distinct after evaluation and there may be at most one `default:` label inside a `switch`.

E 103: may not subtract pointer from scalar
The only operands allowed on subtraction of pointers is pointer - pointer, or pointer - scalar. So, scalar - pointer is not allowed. The following is an example of this error:

```
int i;
int *pi = &i;
ff(1 - pi);             /* error */
```

E 104: left operand of *operator* has not struct/union type
The first operand of a '.' or '->' must have a `struct` or `union` type.

E 105: zero or negative array size - ignored
Array bound constants must be greater than zero. So, `char a[0];` is not allowed.

E 106: different constructors
Compatible function types with parameter type lists must agree in number of parameters and in use of ellipsis. Also, the corresponding parameters must have compatible types. This error is usually followed by informational message I 111. The following is an example of this error:

```
int f(int);
int f(int, int);        /* error different parameter list */
```

E 107: different array sizes
Corresponding array parameters of compatible function types must have the same size. This error is usually followed by informational message I 111.

The following is an example of this error:

```
int f(int [][2]);
int f(int [][3]);       /* error */
```

E 108: different types

Corresponding parameters must have compatible types and the type of each prototype parameter must be compatible with the widened definition parameter. This error is usually followed by informational message I 111. The following is an example of this error:

```
int f(int);
int f(long);            /* error different type in parameter list */
```

E 109: floating point constant out of valid range

A floating point constant must have a value that fits in the type to which it was assigned. See Section 1.2.3, "Data Types", for the valid range of a floating point constant. The following is an example of this error:

```
float d = 10E9999;      /* error, too big */
```

E 110: function cannot return arrays or functions

A function may not have a return type that is of type array or function. A pointer to a function is allowed. The following are examples of this error:

```
typedef int F(); F f();         /* error */
typedef int A[2]; A g();        /* error */
```

I 111: parameter list does not match earlier prototype

Check the parameter list or adjust the prototype. The number and type of parameters must match. This message is preceded by error E 106, E 107 or E 108.

E 112: parameter declaration must include identifier

If the declarator is a prototype, the declaration of each parameter must include an identifier. Also, an identifier declared as a `typedef` name cannot be a parameter name. The following are examples of this error:

```
int f(int g, int) {return (g);} /* error */
```

```
typedef int int_type;
int h(int_type) {return (0);}   /* error */
```

E 114: incomplete struct/union type

The `struct` or `union` type must be known before you can use it. The following is an example of this error:

```
extern struct unknown sa, sb;
sa = sb;        /* 'unknown' does not have a defined type */
```

The left side of an assignment (the lvalue) must be modifiable.

E 115: label "*name*" undefined

A `goto` statement was found, but the specified label did not exist in the same function or module. The following is an example of this error:

```
f1() { a: ; }     /* W 116 */
f2() { goto a; } /* error, label 'a:' is not defined in f2() */
```

W 116: label "*name*" not referenced

The given label was defined but never referenced. The reference of the label must be within the same function or module. The following is an example of this warning:

```
f() { a: ; }    /* 'a' is not referenced */
```

E 117: "*name*" undefined

The specified identifier was not defined. A variable's type must be specified in a declaration before it can be used. This error can also be the result of a previous error. The following is an example of this error:

```
unknown i;      /* error, 'unknown' undefined */
i = 1;          /* as a result, 'i' is also undefined */
```

W 118: constant expression out of valid range
A constant expression used in a `case` label may not be too large. Also when converting a floating point value to an integer, the floating point constant may not be too large. This warning is usually preceded by error E 16 or E 109. The following is an example of this warning:

```
int i = 10E88;   /* error and warning */
```

E 119: cannot take 'sizeof' bitfield or void type
The size of a bit field or `void` type is not known. So, the size of it cannot be taken.

E 120: cannot take 'sizeof' function
The size of a function is not known. So, the size of it cannot be taken.

E 121: not a function declarator
This is not a valid function. This may be due to a previous error. The following is an example of this error:

```
int f() return 0;       /* missing '{ }' */
int g() { }             /* error, 'g' is not a formal parameter and
                           therefore, this is not a valid function
                           declaration */
```

E 122: unnamed formal parameter
The parameter must have a valid name.

W 123: function should return something
A return in a non-`void` function must have an expression.

E 124: array cannot hold functions
An array of functions is not allowed.

E 125: function cannot return anything
A `return` with an expression may not appear in a `void` function.

W 126: missing return (function "*name*")
A non-`void` function with a non-empty function body must have a `return` statement.

E 129: cannot initialize "*name*"
Declarators in the declarator list may not contain initializations. Also, an `extern` declaration may have no initializer. The following are examples of this error:

```
{ extern int i = 0; }  /* error */
int f( i ) int i=0;    /* error */
```

W 130: operands of *operator* are pointers to different types
Pointer operands of an operator or assignment ('='), must have the same type. For example, the following code generates this warning:

```
long *pl;
int  *pi = 0;
pl = pi;        /* warning */
```

E 131: bad operand type(s) of *operator*
The operator needs an operand of another type. The following is an example of this error:

```
int *pi;
pi += 1.;       /* error, pointer on left; needs
                          integral value on right */
```

W 132: value of variable "*name*" is undefined
This warning occurs if a variable is used before it is defined. For example, the following code generates this warning:

```
int a,b;
a = b;          /* warning, value of b unknown */
```

E 133: illegal struct/union member type
A function cannot be a member of a `struct` or `union`. Also, bit fields may only have type `int` or `unsigned`.

E 134: bitfield size out of range - set to 1
The bit field width may not be greater than the number of bits in the type and may not be negative. The following example generates this error:

```
struct i { unsigned i : 999; }; /* error */
```

W 135: statement not reached
The specified statement will never be executed. This is for example the case when statements are present after a `return`.

E 138: illegal function call
You cannot perform a function call on an object that is not a function. The following example generates this error:

```
int i, j;
j = i();        /* error, i is not a function */
```

E 139: *operator* cannot have aggregate type
The type name in a (cast) must be a scalar (not a `struct`, `union` or a pointer) and also the operand of a (cast) must be a scalar. The following are examples of this error:

```
static union ui {int a;} ui ;
ui = (union ui)9;       /* cannot cast to union */
ff( (int)ui );          /* cannot cast a union to something else */
```

E 140: *type* cannot be applied to a register/bit/bitfield object or builtin/inline function
For example, the '&' operator (address) cannot be used on registers and bit fields. So, `func(&r6);` and `func(&bitf.a);` are invalid.

E 141: *operator* requires modifiable lvalue
The operand of the '++', or '--' operator and the left operand of an assignment or compound assignment (lvalue) must be modifiable. The following is an example of this error:

```
const int i = 1;
i = 3;                  /* error, const cannot be modified */
```

E 143: too many initializers
There may be no more initializers than there are objects. The following is an example of this error:

```
static int a[1] = {1, 2};       /* error, only one object
                                          can be initialized */
```

W 144: enumerator "*name*" value out of range
An `enum` constant exceeded the limit for an `int`. The following is an example of this warning:

```
enum { A = INT_MAX, B };        /* warning, B does not fit
                                          in an int anymore */
```

E 145: requires enclosing curly braces
A complex initializer needs enclosing curly braces. For example, `int a[] = 2;` is not valid, but `int a[] = {2};` is.

E 146: argument #*number*: memory spaces do not match
With prototypes, the memory spaces of arguments must match.

W 147: argument #*number*: different levels of indirection
With prototypes, the types of arguments must be assignment compatible. The following code generates this warning:

```
int i; void func(int,int);
func( 1, &i );          /* warning, argument 2 */
```

W 148: argument #*number*: struct/union type does not match
With prototypes, both the prototyped function argument and the actual argument was a `struct` or `union`, but they have different tags. The tag types should match.
The following is an example of this warning:

```
f(struct s);            /* prototype */
main()
{
   struct { int i; } t;
   f( t );              /* t has other type than s */
}
```

E 149: object "*name*" has zero size
A `struct` or `union` may not have a member with an incomplete type. The following is an example of this error:

```
struct { struct unknown m; } s;        /* error */
```

W 150: argument #*number*: pointers to different types
With prototypes, the pointer types of arguments must be compatible. The following example generates this warning:

```
int f(int*);
long *l;
f(l);                   /* warning */
```

W 151: ignoring memory specifier
Memory specifiers for a `struct`, `union` or `enum` are ignored.

E 152: operands of *operator* are not pointing to the same memory space
Be sure the operands point to the same memory space. This error occurs, for example, when you try to assign a pointer to a pointer from a different memory space.

E 153: 'sizeof' zero sized object
An implicit or explicit `sizeof` operation references an object with an unknown size. This error is usually preceded by error E 119 or E 120, cannot take 'sizeof'.

E 154: argument #*number*: struct/union mismatch
With prototypes, only one of the prototyped function argument or the actual argument was a `struct` or `union`. The types should match. The following is an example of this error:

```
f(struct s);    /* prototype */

main()
{
   int i;
   f( i );      /* i is not a struct */
}
```

E 155: casting lvalue '*type*' to '*type*' is not allowed
The operand of the '++', or '--' operator or the left operand of an assignment or compound assignment (lvalue) may not be cast to another type.
The following is an example of this error:

```
int i = 3;
++(unsigned)i;  /* error, cast expression is not an lvalue */
```

E 157: "*name*" is not a formal parameter
If a declarator has an identifier list, only its identifiers may appear in the declarator list. The following is an example of this error:

```
int f( i ) int a;       /* error */
```

E 158: right side of *operator* is not a member of the designated struct/union
The second operand of '.' or '->' must be a member of the designated `struct` or `union`.

E 160: pointer mismatch at *operator*
Both operands of *operator* must be a valid pointer. The following example generates this error:

```
int *pi = 44;   /* right side not a pointer */
```

E 161: aggregates around *operator* do not match
The contents of the structs, unions or arrays on both sides of the *operator* must be the same. The following example causes this error:

```
struct {int a; int b;} s;
struct {int c; int d; int e;} t;
s = t;          /* error */
```

E 162: *operato*r requires an lvalue or function designator
The '&' (address) operator requires an lvalue or function designator. The following is an example of this error:

```
int i;
i = &( i = 0 );
```

W 163: operands of *operator* have different level of indirection
The types of pointers or addresses of the operator must be assignment compatible. The following is an example of this warning:

```
char **a;
char *b;
a = b;          /* warning */
```

E 164: operands of *operator* may not have type 'pointer to void'
The operands of *operator* may not have operand (`void *`).

W 165: operands of *operator* are incompatible: pointer vs. pointer to array
The types of pointers or addresses of the operator must be assignment compatible. A pointer cannot be assigned to a pointer to array. The following is an example of this warning:

```
main()
{
        typedef int array[10];
        array a;
        array *ap = a;  /* warning */
}
```

E 166: *operator* cannot make something out of nothing
Casting type `void` to something else is not allowed. The following example generates this error:

```
void f(void);
main()
{
        int i;
        i = (int)f();   /* error */
}
```

E 170: recursive expansion of inline function "*name*"
An `_inline` function may not be recursive. The following example generates this error:

```
_inline int a (int i)
{
        a(i);           /* recursive call */
        return i;
}
main()
{
        a(1);           /* error */
}
```

E 171: too much tail-recursion in inline function "*name*"
If the function level is greater than or equal to 40 this error is given. The following example generates this error:

```
_inline void a ()
{
        a();
}
main()
{
        a();
}
```

W 172: adjacent strings have different types
When concatenating two strings, they must have the same type. The following example generates this warning:

```
char b[] = L"abc""def";/* strings have different types */
```

E 173: 'void' function argument
A function may not have an argument with type `void`.

E 174: not an address constant
A constant address was expected. Unlike a static variable, an automatic variable does not have a fixed memory location and therefore, the address of an automatic is not a constant. The following is an example of this error:

```
int *a;
static int *b = a;      /* error */
```

E 175: not an arithmetic constant
In a constant expression no assignment operators, no '++' operator, no '--' operator and no functions are allowed. The following is an example of this error:

```
int a;
static int b = a++;     /* error */
```

E 176: address of automatic is not a constant
Unlike a static variable, an automatic variable does not have a fixed memory location and therefore, the address of an automatic is not a constant. The following is an example of this error:

```
int a;                  /* automatic */
static int *b = &a;     /* error */
```

W 177: static variable "*name*" not used
A static variable is declared which is never used. To eliminate this warning remove the unused variable.

W 178: static function "*name*" not used
A static function is declared which is never called. To eliminate this warning remove the unused function.

E 179: inline function "*name*" is not defined
Possibly only the prototype of the inline function was present, but the actual inline function was not. The following is an example of this error:

```
_inline int a(void);    /* prototype */
main()
{
        int b;
        b = a();        /* error */
};
```

E 180: illegal target memory (*memory*) for pointer
The pointer may not point to *memory*. For example, a pointer to bitaddressable memory is not allowed.

W 182: argument #*number*: different types
With prototypes, the types of arguments must be compatible.

I 185: (prototype synthesized at line *number* in "*name*")
This is an informational message containing the source file position where an old-style prototype was synthesized. This message is preceded by error E 146, W 147, W 148, W 150, E 154, W 182 or E 203.

E 186: array of type bit is not allowed
An array cannot contain bit type variables.

E 187: illegal structure definition
A structure can only be defined (initialized) if its members are known.
So, `struct unknown s = { 0 };` is not allowed.

E 188: structure containing bit-type fields is forced into bitaddressable area
This error occurs when you use a bitaddressable storage type for a structure containing bit-type members.

E 189: pointer is forced to bitaddressable, pointer to bitaddressable is illegal
A pointer to bitaddressable memory is not allowed.

W 190: "long float" changed to "float"
In ANSI C floating point constants are treated having type `double`, unless the constant has the suffix 'f'. If you have specified an option to use float constants, a `long` floating point constant such as `123.12fl` is changed to a `float`.

E 191: recursive struct/union definition
A `struct` or `union` cannot contain itself. The following example generates this error:

```
struct s { struct s a; } b;    /* error */
```

E 192: missing filename after -f option
The **-f** option requires a filename argument.

E 194: cannot initialize typedef
You cannot assign a value to a typedef variable. So, `typedef i=2;` is not allowed.

F 199: demonstration package limits exceeded
The demonstration package has certain limits which are not present in the full version. Contact Seiko Epson for a full version.

W 200: unknown pragma - ignored
The compiler ignores pragmas that are not known. For example, `#pragma unknown`.

W 201: *name* cannot have storage type - ignored
A `register` variable or an automatic/parameter cannot have a storage type. To eliminate this warning, remove the storage type or place the variable outside a function.

E 202: '*name*' is declared with 'void' parameter list
You cannot call a function with an argument when the function does not accept any (`void` parameter list). The following is an example of this error:

```
int f(void);            /* void parameter list */

main()
{
        int i;
        i = f(i);       /* error */
        i = f();        /* OK */
}
```

E 203: too many/few actual parameters
With prototyping, the number of arguments of a function must agree with the prototype of the function. The following is an example of this error:

```
int f(int);             /* one parameter */

main()
{
        int i;
        i = f(i,i);     /* error, one too many */
        i = f(i);       /* OK */
}
```

W 204: U suffix not allowed on floating constant - ignored
A floating point constant cannot have a 'U' or 'u' suffix.

W 205: F suffix not allowed on integer constant - ignored
An integer constant cannot have a 'F' or 'f' suffix.

E 206: '*name*' named bit-field cannot have 0 width
A bit field must be an integral constant expression with a value greater than zero.

E 212: "*name*": missing static function definition
A function with a `static` prototype misses its definition.

W 303: variable '*name*' uninitialized
Possibly an initialization statement is not reached, while a function should return something. The following is an example of this warning:

```
int a;

int f(void)
{
   int i;

   if ( a )
   {
        i = 0;  /* statement not reached */
   }
   return i;    /* warning */
}
```

E 327: too many arguments to pass in registers for _asmfunc '*name*'
An `_asmfunc` function uses a fixed register-based interface between C and assembly, but the number of arguments that can be passed is limited by the number of available registers. With function *name* this limit was reached.

Backend

W 501: function qualifier used on non-function
A function qualifier can only be used on functions.

E 502: Intrinsic function '_int()' needs an immediate value as parameter
The argument of the `_int()` intrinsic function must be an integral constant expression rather than any type of integral expression.

E 503: Intrinsic function '_jrsf()' needs an immediate value 0..3
The given number must be a constant value between 0 and 3.

W 508: function qualifier duplicated
Only one function qualifier is allowed.

E 511: interrupt function must have void result and void parameter list
A function declared with `_interrupt(n)` may not accept any arguments and may not return anything.

W 512: '*number*' illegal interrupt number (0, or 3 to 251) - ignored
The interrupt vector number must be 0, or in the range 3 to 251. Any other number is illegal.

E 513: calling an interrupt routine, use '_swi()'
An interrupt function cannot be called directly, you must use the intrinsic function `_swi()`.

E 514: conflict in '_interrupt'/'_asmfunc' attribute
The attributes of the current function qualifier declaration and the previous function qualifier declaration are not the same.

E 515: different '_interrupt' number
The interrupt number of the current function qualifier declaration and the previous function qualifier declaration are not the same.

E 516: '*memory_type*' is illegal memory for function
The storage type is not valid for this function.

W 517: conversion of long address to short address
This warning is issued when pointer conversion is needed, for example, when you assign a `_huge` pointer to a `_near` pointer.

F 524: illegal memory model
See the compiler usage for valid arguments of the **-M** option.

E 526: function qualifier '_asmfunc' not allowed in function definition
`_asmfunc` is only allowed in the function prototype.

E 528: _at() requires a numerical address
You can only use an expression that evaluates to a numerical address.

E 529: _at() address out of range for this type of object
The absolute address is not present in the specified memory space.

E 530: _at() only valid for global variables
Only global variables can be placed on absolute addresses.

E 531: _at() only allowed for uninitialized variables
Absolute variables cannot be initialized.

E 532: _at() has no effect on external declaration
When declared `extern` the variable is not allocated by the compiler.

W 533: c88 language extension keyword used as identifier
A language extension keyword is a reserved word, and reserved words cannot be used as an identifier.

E 536: illegal syntax used for default section name '*name*' in -R option
See the description of the **-R** option for the correct syntax.

E 537: default section name '*name*' not allowed
See the description of the **-R** option for the correct syntax.

W 538: default section name '*name*' already renamed to '*new_name*'
Only use one of the **-R** option or the **renamesect** pragma or use another name.

W 542: optimization stack underflow, no optimization options are saved with #pragma optimize
This warning occurs if you use a `#pragma endoptimize` while there were no options saved by a previous `#pragma optimize`.

W 555: current optimization level could reduce debugging comfort (-g)
You could have HLL debug conflicts with these optimization settings.

E 560: Float/Double: not yet implemented
Floating point will be supported in a following version.

APPENDIX B ASSEMBLER ERROR MESSAGES

The assembler produces error messages on standard error output. If the list option of the assembler is effective, error messages will be included in the list file as well, when the assembler has started list file generation. Error messages have the following layout:

[E | F | W] *error_number*: *filename* line *number* : *error_message*

Example:

as88 E214: \tmp\tst.src line 17 : illegal addressing mode

The example reports the error, starting with the severity (E: error, F: fatal error, W: warning) and the error number followed by the source filename and the line number. The last part of the line shows the error message text.

All warnings (W), errors (E), and fatal errors (F) of **as88** are described below.

WARNINGS (W)

The assembler may generate the following warnings:

W 101: use *option* at the start of the source; ignored
Primary options must be used at the start of the source.

W 102: duplicate attribute "*attribute*" found
An attribute of an EXTERN directive is used twice or more. Remove one of the duplicate attributes.

W 104: expected an attribute but got *attribute*; ignored

W 105: section activation expected, use *name* directive
Use the SECT directive to activate a section.

W 106: conflicting attributes specified "*attributes*"
You used two conflicting attributes in an EXTERN statement directive. For example EXTERN and INTERN. Choose which one you want to use and remove the other.

W 107: memory conflict on object "*name*"
A label or other object is explicit or implicit defined using incompatible memory types. Check all usages and definitions of the object *name* to remove this conflict.

W 108: object attributes redefinition "*attributes*"
A label or other object is explicit or implicit defined using incompatible attributes. For example INTERN and EXTERN. Check all usages and definitions of the object to remove the conflict.

W 109: label "*label*" not used
The label *label* is defined with the GLOBAL directive and neither defined nor referred, or the label is defined with the LOCAL directive and not referenced. You can remove this label and its definitions (in the case of a LOCAL label).

W 110: extern label "*label*" defined in module, made global
The label *label* is defined with an EXTERN directive and defined as a label in the source. The label will be handled as a global label. Change the EXTERN definition into GLOBAL or one of the identifiers.

W 111: unknown $LIST control flag "*flag*"
You supplied an unknown *flag* to the $LIST control. See the description of the $LIST control for the possible arguments.

W 112: text found after END; ignored
An END directive designates the end of the source file. All text after the END directive will be ignored. Remove the text.

W 113: unknown $MODEL specifier; ignored
You supplied an unknown model. See the description of the $MODEL control for all possible models.

W 114: $MODEL may only be specified once, it remains "*model*"; ignored
You supplied more than one model. See the description of the $MODEL control for all possible models.

W 115: use ON or OFF after control name
The control you specified must have either ON or OFF after the control name. See the description of the control for details.

W 116: unknown parameter "*parameter*" for *control-name* control
See the description of the control for the allowed parameters.

W 118: inserted "extern *name*"
The symbol *name* is used inside an expression, but not defined with an EXTERN directive. The assembler inserts an EXTERN definition of the offending symbol. Add an EXTERN definition.

W 119: "*name*" section has not the MAX attribute; ignoring RESET

W 120: assembler debug information: cannot emit non-tiof expression for *label*
The SYMB record contains an expression with operations that are not supported by the IEEE-695 object format. When the SYMB record is generated by the C compiler, please fill out the error report and send it to Seiko Epson.

W 121: changed alignment size to *size*

W 123: expression: *type-error*
The expression performs an illegal operation on an address or combines incompatible memory spaces. Check the expression, and change it.

W 124: cannot purge macro during its own definition

W 125: "*symbol*" is not a defined symbol
You tried to UNDEF a symbol that was not previously DEFINEd or was already undefined. Check all DEFINE/UNDEF combinations of the offending symbol.

W 126: redefinition of "*define-symbol*"
The symbol is already DEFINEd in the current scope. The symbol is redefined according to this DEFINE. UNDEF any symbol before redefining it.

W 127: redefinition of macro "*macro*"
The macro is already defined. The macro is redefined according to this macro definition. Purge any macro using PMACRO before redefining it.

W 128: number of macro arguments is less than definition
You supplied less arguments to the macro than when defining it. Check your macro definition with this macro call. The undefined macro arguments are left empty (as in `DEFINE def ''`).

W 129: number of macro arguments is greater than definition
You supplied more arguments to the macro than when defining it. Check your macro definition with this macro call. The superfluous macro arguments are ignored.

W 130: DUPA needs at least one value argument
The DUPA directive needs at least two arguments, the dummy parameter and a value parameter. Add one or more value-parameters.

W 131: DUPF increment value gives empty macro
The step value supplied with the DUPF macro will skip the DUPF macro body. Check the step value.

W 132: IF started in previous file "*file*", line *line*
The ENDIF or ELSE pre-processor directive matches with an IF directive in another file. Check on any missing ENDIF or ELSE directives in that file.

W 133: currently no macro expansion active
The @CNT() and @ARG() functions can only be used inside a macro expansion. Check your macro definitions or expression.

W 134: "*directive*" is not supported, skipped
The supplied directive is not supported by this assembler. Remove all uses of this directive.

W 135: define symbol of "*define-symbol*" is not an identifier; skipped definition
You supplied an illegal identifier with the **-D** option on the command line. An identifier should start with a letter, followed by any number of letters, digits or underscores.

W 137: label "*label*" defined *attribute* and *attribute*
The label is defined with an EXTERN and a GLOBAL directive. The EXTERN directive is removed, leaving the label global.

W 138: warning: *WARN-directive-arguments*
Output from the WARN directive.

W 139: expression must be between *hex-value* and *hex-value*

W 140: expression must be between *value* and *value*

W 141: *global/local* label "*name*" not defined in this module; made extern
The label is declared and used but not defined in the source file. Check the current scope of the label and its usage, change the declaration to EXTERN or add a label definition.

W 170: code address maps to zero page
The code offset you specified to the @CPAG function is in the zero page.

W 171: address offset must be between 0 and FFFF
The offset you specified in the @CADDR or @DADDR function was too large. The offset must be between 0 and 0FFFFh.

W 172: page number must be between 0 and FF
The page number you specified in the @CADDR or @DADDR function was too large. The page number must be between 0 and 0FFh.

ERRORS (E)

The assembler generates the following error messages when a user error situation occurs. These errors do not terminate assembly immediate. If one or more of these errors occur, assembly stops at the end of the active pass.

E 200: *message*; halting assembly
The assembler stops the further processing of your source file. This is only an informative message. Remove all errors reported earlier and try again.

E 201: unexpected newline or line delimiter
The syntax checker found a newline or line delimiter that does not confirm to the assembler grammar. Check the line for syntax errors or remove the offending newline or line delimiter.

E 202: unexpected character: '*character*'
The syntax checker found a character that does not confirm to the assembler grammar. Check the line for syntax errors or remove the offending character.

E 203: illegal escape character in string constant
The syntax checker found an illegal escape character in the string constant that does not confirm to the assembler grammar. Check the line for syntax errors or remove the offending escape character.

E 204: I/O error: open intermediate file failed (*file*)
The assembler opens an intermediate file to optimize the lexical scanning phase. The assembler cannot open this file. The assembler checks if the environment symbol TMPDIR is set. If so, this directory is used for opening the file. Otherwise the file is opened in the current directory.

E 205: syntax error: expected *token* at *token*
The syntax checker expected to find a token but found another token. The expected token is inserted instead of the found token. Check the line for syntax errors.

E 206: syntax error: *token* unexpected
The syntax checker found an unexpected token. The offending token is removed from the input and assembling continues. Check the line for syntax errors.

E 207: syntax error: missing ':'
The syntax checker found a label definition or memory space modifier but missed the appended semi-colon. Check the line for syntax errors, for example misspelled mnemonics.

E 208: syntax error: missing ')'
The syntax checker expected to find a closing parentheses. Check the expression syntax for missing operators and nesting of parentheses.

E 209: invalid radix value, should be 2, 8, 10 or 16
The RADIX directive accepts only 2, 8, 10 or 16.

E 210: syntax error
The syntax checker found an error. Check the line for syntax errors.

E 211: unknown model
Substitute the correct model, one of s, c, d or l.

E 212: syntax error: expected *token*
The syntax checker expected to find a token but found nothing. The expected token is inserted. Check the line for syntax errors.

E 213: label "*label*" defined *attribute* and *attribute*
The label is defined with a LOCAL and a GLOBAL or EXTERN directive. Check your label scoping or change the label declarations.

E 214: illegal addressing mode
The mnemonic used an illegal addressing mode. Check the register usage of address constructs.

E 215: not enough operands
The mnemonic needs more operands. Check the source line and change the instruction.

E 216: too many operands
The mnemonic needs less operands. Check the source line and change the instruction.

E 217: *description*
There was an error found during assembly of the mnemonic. Check the instruction.

E 218: unknown mnemonic: "*name*"
The assembler found an unknown mnemonic. Check the instruction. It could be that you specified a label but forgot the ':'.

E 219: this is not a hardware instruction (use $OPTIMIZE OFF "H")
The assembler found a generic instruction, but the **-Oh** (hardware only) option or the $OPTIMIZE ON "H" control was specified.

E 223: unknown section "*name*"
The section name specified with a SECT directive has not (yet) been defined with a DEFSECT directive. Check the SECT name and the corresponding DEFSECT name.

E 224: unknown label "*name*"
A label was used which was not defined. Check that the label and its definition have the same name.

E 225: invalid memory type
You supplied an invalid memory modifier.

E 226: unknown symbol attribute: *attribute*

E 227: invalid memory attribute
The assembler found an unknown location counter or memory mapping attribute.

E 228: *attr* attribute needs a number
The attribute *attr* needs an extra parameter. For example, the FIT attribute.

E 229: only one of the *name* attributes may be specified

E 230: invalid section attribute: *name*
The assembler found an unknown section attribute.

E 231: absolute section, expected "AT" expression
An absolute section must be specified using an 'AT *address*' expression.

E 232: MAX/OVERLAY sections need to be named sections
Sections with the MAX or OVERLAY attribute must have a name, otherwise the locator cannot overlay the sections.

E 233: *type* section cannot have *attribute* attribute
Code sections may not have the CLEAR or OVERLAY attribute.

E 234: section attributes do not match earlier declaration
In an previous definition of the same section other attributes were used. Check all section definitions with the same name.

E 235: redefinition of section
An absolute section of the same name can only be located once.

E 236: cannot evaluate expression of *descriptor*
Some functions and directives must evaluate their arguments during assembly. Change the expression so that it can be evaluated. It could have cyclic dependencies on symbol locations.

E 237: *descriptor* directive must have positive value
Some directives need to have a positive argument. Check the expression so that is evaluates to a positive number.

E 238: Floating point numbers not allowed with DB directive
The DB directive does not accept floating point numbers. Convert the expressions or use the DW directive instead.

E 239: byte constant out of range
The DB directive stores expressions in bytes. A byte can only hold numbers between 0 and 255.

E 240: word constant out of range
The DW directive stores expressions in words. A word can hold 16 bit numbers. Check the range of the expression.

E 241: Cannot emit non tiof functions, replaced with integral value '0'
Floating point expressions and some functions can not be represented in the IEEE-695 object format. When an expression contains unknown symbols it cannot be evaluated and not emitted to the object file. Change these expressions to integral expressions, or make sure they can be evaluated during assembly.

E 242: the *name* attribute must be specified
A section must have the CODE or DATA attribute.

E 243: use $OBJECT OFF or $OBJECT "*object-file*"

E 244: unknown control "*name*"
The specified control does not exist. See Section 2.7, "Assembler Controls", for a description of all available controls.

E 246: ENDM within IF/ENDIF
The assembler found an ENDM directive within an IF/ENDIF pair. Check the macro and DUP definitions or remove this directive.

E 247: illegal condition code
The assembler encountered an illegal condition code within an instruction. Check your input line.

E 248: cannot evaluate origin expression of org "*name*: *address*"
All origins of absolute sections must be evaluated before creation of the object file. Check the address expression on the usage of undefined or location dependant symbols.

E 249: incorrect argument types for function "*function*"
The supplied argument(s) evaluated to a different type than expected. Change the argument expressions to the correct type.

E 250: tiof function not yet implemented: "*function*"
The supplied object format function is not yet implemented.

E 251: @POS(,,*start*) start argument past end of string
The *start* argument is larger than the length of the string in the first parameter. Change *start* to the correct range.

E 252: second definition of label "*label*"
The label is defined twice in the same scope. Check the label definitions and rename of remove duplicate definitions.

E 253: recursive definition of symbol "*symbol*"
The evaluation of the symbol depends on its own value. Change the symbol value exclude this cyclic definition.

E 254: missing closing '>' in include directive
The syntax checker missed the closing '>' bracket in the include directive. Add a closing '>'.

E 255: could not open include file *include-file*
The assembler could not open the given include-file. Check the current search path for the presence of the include file and if it may be read.

E 256: integral divide by zero
The expression contains an divide by zero. This is not defined. Change the expression to exclude a division by zero.

E 257: unterminated string
All strings must end on the same line as they are started. Check for a missing ending quote.

E 258: unexpected characters after macro parameters, possible illegal white space
Spaces are not permitted between macro parameters. Check the syntax of the macro call.

E 259: COMMENT directive not permitted within a macro definition and conditional assembly
This assembler does not permit the usage of the COMMENT directive within MACRO/DUP definitions or IF/ELSE/ENDIF constructs. Replace the offending COMMENTs with comments starting with a semicolon.

E 260: definition of "*macro*" unterminated, missing "endm"
The macro definition is not terminated with an ENDM directive. Check the macro definition.

E 261: macro argument name may not start with an '_'
MACRO and DUP arguments may not start with an underscore. Replace the offending parameter names with non-underscore names.

E 262: cannot find "*symbol*"
Could not find a definition of the argument of a '%' or '?' operator within a macro expansion. Check for a definition of the offending symbol.

E 263: cannot evaluate: "*symbol*", value is unknown at this point
The symbol used with a '%' or '?' operator within a macro expansion has not been defined. Insert a definition of the offending identifier.

E 264: cannot evaluate: "*symbol*", value depends on an unknown symbol
Could not evaluate the argument of a '%' or '?' operator within a macro expansion. Check the definition of the offending symbol.

E 265: cannot evaluate argument of *dup* (unknown or location dependant symbols)
The arguments of the DUP directive could not be evaluated. Check the argument expressions on forward references or unknown symbols.

E 266: *dup* argument must be integral
The argument of the DUP directive must be integral. Change the expression so that it evaluates to an integral number.

E 267: *dup* needs a parameter
Check the syntax of the DUP directive.

E 268: ENDM without a corresponding MACRO or DUP definition
The assembler found an ENDM directive without an corresponding MACRO or DUP definition. Check the macro and dup definitions or remove this directive.

E 269: ELSE without a corresponding IF
The assembler found an ELSE directive without an corresponding IF directive. Check the IF/ELSE/ENDIF nesting or remove this directive.

E 270: ENDIF without a corresponding IF
The assembler found an ENDIF directive without an corresponding IF directive. Check the IF/ELSE/ENDIF nesting or remove this directive.

E 271: missing corresponding ENDIF
The assembler found an IF or ELSE directive without an corresponding ENDIF directive. Check the IF/ELSE/ENDIF nesting or remove this directive.

E 272: label not permitted with this directive
Some directives do not accept labels. Move the label to a line before or after this line.

E 273: wrong number of arguments for *function*
The function needs more or less arguments. Check the function definition and add or remove arguments.

E 274: illegal argument for *function*
An argument has the wrong type. Check the function definition and change the arguments accordingly.

E 275: expression not properly aligned

E 276: immediate value must be between *value* and *value*
The immediate operand of the instruction does only accept values in the given range. Use the '&' operator to force a value within the needed range or use '#>' to force a long immediate operand.

E 277: address must be between $*address* and $*address*
The address operand is not in the range mentioned. Change the address expression.

E 278: operand must be an address
The operand must be an address but has no address attributes. Use an address modifier or change the address expression.

E 279: address must be short

E 280: address must be short
The operand must be an address in the short range. The expression evaluated to a long address or an address in an unknown range.

E 281: illegal option "*option*"
The assembler found an unknown or misspelled command line option. The option will be ignored.

E 282: "Symbols:" part not found in map file "*name*"
The map file may be incomplete. Check if it is correctly produced by the locator.

E 283: "Sections:" part not found in map file "*name*"
The map file may be incomplete. Check if it is correctly produced by the locator.

E 284: module "*name*" not found in map file "*name*"
The map file may be incomplete. Check if it is correctly produced by the locator.

E 285: *file-kind* file will overwrite *file-kind* file
The assembler warns when one of its output files will overwrite the source file you gave on the command line or another output file. Change the name of the source file, use the **-o** option to change the name of the output file or remove the **-err** option to suppress the generation of the error file.

E 286: $CASE options must be given before any symbol definition
The $CASE options may only be given before any symbol is defined. Move the options to the start of the first source file.

E 287: symbolic debug error: *message*
The assembler found an error in a symbolic debug (SYMB) instruction. When the SYMB instruction is generated by the C compiler, please fill out the error report form and send it to Seiko Epson. As a work around you could disable the symbolic debug information of this module (remove the **-g** option).

E 288: error in PAGE directive: *message*
The arguments supplied to the PAGE directive do not conform to the restrictions. Check the PAGE directive restrictions in the manual and change the arguments accordingly.

E 290: fail: *message*
Output of the FAIL directive. This is an user generated error. Check the source code to see why this FAIL directive is executed.

E 291: generated check: *message*
Integrity check for the coupling between the C compiler and assembler. You should not see this error message, unless there are errors in user inserted assembly (using the "#pragma asm" construct).

E 293: expression out of range
An instruction operand must be in a specified address range. Check the address expression, change it.

E 294: expression must be between *hexvalue* and *hexvalue*

E 295: expression must be between *value* and *value*

E 296: optimizer error: *message*
The optimizer found an error. Try to change the instruction or turn off the optimizer.

E 297: jump address must be a code address
Jumps and jump-subroutines must have a target address in code memory. Check the address expression or use a memory modifier to force the expression into code memory.

E 298: size depends on location, cannot evaluate
The size of some constructions (notably the align directives) depend on the memory address. Change the offending construction.

FATAL ERRORS (F)

The following errors cause the assembler to terminate immediately. Fatal errors are usually due to user errors.

F 401: memory allocation error
A request for free memory is denied by the system. All memory has been used. You may have to break your program down into smaller pieces.

F 402: duplicate input filename "*file*" and "*file*"
The assembler requires one input filename on the command line. Two or more filenames is erroneous.

F 403: error opening *file-kind* file: "*file-name*"
The assembler could not open the given file. When this is a source file, check if the file you specified at the command line exists and if it is readable. When the file is a temporary file, check if the environment symbol TMPDIR has been set correctly.

F 404: protection error: *message*
No protection key or not a IBM compatible PC.

F 405: I/O error
The assembler cannot write its output to a file. Check if you have enough free disk space.

F 406: parser stack overflow

F 407: symbolic debug output error
The symbolic debug information is incorrectly written in the object file. Please fill out the error report form and send it to Seiko Epson.

F 408: illegal operator precedence
The operator priority table is corrupt. Please fill out the error report form and send it to Seiko Epson.

F 409: Assembler internal error
The assembler encountered internal inconsistencies. Please fill out the error report form and send it to Seiko Epson.

F 410: Assembler internal error: duplicate mufom "*symbol*" during rename
The assembler renames all symbols local to a scope to unique symbols. In this case the assembler did not succeed into making an unique name. Please fill out the error report form and send it to Seiko Epson.

F 411: symbolic debug error: "*message*"
An error occurred during the parsing of the SYMB directive. When this SYMB directive is generated by the C compiler, please fill out the error report form and send it to Seiko Epson.

F 412: macro calls nested too deep (possible endless recursive call)
There is a limit to the number of nested macro expansions. Currently this limit is set to 1000. Check for recursive definitions or try to simplify your source when you encounter this restriction.

F 413: cannot evaluate "*function*"
A function call is encountered although it should have been processed. As a work-around, try to locate the offending function call and remove it from your source. Please fill out the error report form and send it to Seiko Epson.

F 414: cannot recover from previous errors, stopped
Due to earlier errors the assembler internal state got corrupted and stops assembling your program. Remove the errors reported earlier and retry.

F 415: error opening temporary file
The assembler uses temporary files for the debug information and list file generation. It could not open or create one of those temporary files. Check if the environment symbol TMPDIR has been set correctly.

F 416: internal error in optimizer
The optimizer found a deadlock situation. Try to assemble without any optimization options. Please fill out the error report form and send it to Seiko Epson.

APPENDIX C LINKER ERROR MESSAGES

Error and warning messages of the linker start with a letter followed by a number and an informational text. The error letter indicates the error type:

W warning
E error
F fatal error
V verbose message

WARNINGS (W)

W 100: Cannot create map file *filename*, turned off -M option
The given file could not be created.

W 101: Illegal filename (*filename*) detected
A filename with an illegal extension was detected.

W 102: Incomplete type specification, type index = *Thexnumber*
An unknown type reference. Arises if a pointer to an unspecified structure is defined.

W 103: Object name (*name*) differs from filename
Internal name of object file not the same as the filename. The file was probably renamed.

W 104: '-o *filename*' option overwrites previous '-o *filename*'
Second **-o** option encountered, previous name is lost.

W 105: No object files found
No files where specified at the invocation.

W 106: No search path for system libraries. Use -L or env "*variable*"
System library files (those given with the **-l** option) must have a search path, either supplied by means of the environment, or by means of the option **-L**.

W 108: Illegal option: *option* (-H or -\? for help)
An illegal option was detected.

W 109: Type not completely specified for symbol <*symbol*> in *file*
Not a complete type specification in either the current file or the mentioned file. This could be an array with unknown depth, or a function with unknown parameters.

W 110: Compatible types, different definitions for symbol <*symbol*> in *file*
Name conflict between compatible types. This could be a member name, tag name for a struct, or a different type name for equal sized basic types (int, long). Note that a basic type conflict is a non portable construct.

W 111: Signed/unsigned conflict for symbol <*symbol*> in *file*
Size of both types is correct, but one of the types contains an unsigned where the other uses a signed type.

W 112: Type conflict for symbol <*symbol*> in *file*
A real type conflict.

W 113: Table of contents of *file* out of date, not searched. (Use ar ts <*name*>)
The **ar** library has a symbol table which is not up to date. Generate a new one with '**ar ts**'.

W 114: No table of contents in *file*, not searched. (Use ar ts <*name*>)
The **ar** library has no symbol table. Generate one with '**ar ts**'.

W 115: Library *library* contains ucode which is not supported
Ucode is not supported by the linker.

W 116: Not all modules are translated with the same threshold (-G value)
The library file has an unknown format, or is corrupted.

W 117: No type found for <*symbol*>. No type check performed
No type has been generated for the symbol.

W 118: Variable <*name*>, has incompatible external addressing modes with file <*filename*>
A variable is not yet allocated but two external references are made by non overlapping addressing modes. This is always an error.

W 119: error from the Embedded Environment: *message*, switched off relaxed addressing mode check
If the embedded environment is readable for the linker, the addressing mode check is relaxed. For instance, a variable defined as data may be accessed as huge. For an overview of the embedded environment error messages, see Appendix F, "Embedded Environment Error Messages".

ERRORS (E)

E 200: Illegal object, assignment of non existing var *var*
The MUFOM variable did not exist. Corrupted object file.

E 201: Bad magic number
The magic number of a supplied library file was not ok.

E 202: Section *name* does not have the same attributes as already linked files
Named section with different attributes encountered. Use **-t** *flag* to see which files are already linked. It is possible that a previously linked file started a .out section with wrong attributes.

E 203: Cannot open *filename*
A given file was not found.

E 204: Illegal reference in address of *name*
Illegal MUFOM variable used in value expression of a variable. Corrupted object file.

E 205: Symbol '*name*' already defined in <*name*>
A symbol was defined twice. The message gives the files involved.

E 206: Illegal object, multi assignment on *var*
The MUFOM variable was assigned more than once probably due to a previous error 'already defined', E 205.

E 207: Object for different processor characteristics
Bits per MAU, MAU per address or endian for this object differs with the first linked object.

E 208: Found unresolved external(s):
There were some symbols not found. If **-r** is not set, this is an error.

E 209: Object format in *file* not supported
The object file has an unknown format, or is corrupted.

E 210: Library format in *file* not supported
The library file has an unknown format, or is corrupted.

E 211: Function <*function*> cannot be added to the already built overlay pool <*name*>
The overlay pool has already been built in a previous linker action. Use option **-r** to prevent this.

E 212: Duplicate absolute section name <*name*>
Absolute sections begin on a fixed address. They cannot be linked.

E 213: Section <*name*> does not have the same size as the already linked one
A section with the EQUAL attribute does not have the same size as other, already linked, sections.

E 214: Missing section address for absolute section <*name*>
Each absolute section must have a section address command in the object. Corrupted object file.

E 215: Section <*name*> has a different address from the already linked one
Two absolute sections may be linked (overlaid) on some conditions. They must have the same address.

E 216: Variable <*name*>, *name* <*name*> has incompatible external addressing modes
A variable is allocated outside a referencing addressing space. For instance, the variable was not allocated in the zero page and this variable was referenced with the zero page addressing mode. This is always an error.

E 217: Variable <*name*>, has incompatible external addressing modes with file <*filename*>
A variable is not yet allocated but two external references are made by non overlapping addressing modes. This is always an error.

E 218: Variable <*name*>, also referenced in <*name*> has an incompatible address format
Addresses are often expressed in bytes. In some special cases, the address is expressed in bits. This is necessary for bit variables. An attempt was made to link different address formats between the current file and the mentioned file.

E 219: Not supported/illegal *feature* in object format *format*
An option/feature is not supported or illegal in given object format.

E 220: page size (0x*hexvalue*) overflow for section <*name*> with size 0x*hexvalue*
Section is too big to fit into the page.

E 221: *message*
Error generated by the object. These errors are in fact generated by the assembler.

E 222: Address of <*name*> not defined
No address was assigned to the variable. Corrupted object file.

FATAL ERRORS (F)

F 400: Cannot create file *filename*
The given file could not be created.

F 401: Illegal object: Unknown command at offset *offset*
An unknown command was detected in the object file. Corrupted object file.

F 402: Illegal object: Corrupted hex number at offset *offset*
Wrong byte count in hex number. Corrupted object file.

F 403: Illegal section index
A section index out of range was detected. Corrupted object file.

F 404: Illegal object: Unknown hex value at offset *offset*
An unknown variable was detected in the object file. Corrupted object file.

F 405: Internal error *number*
Internal fatal error. Passed number will give more information!

F 406: *message*
No key no IBM compatible PC.

F 407: Missing section size for section <*name*>
Each section must have a section size command in the object. Corrupted object file.

F 408: Out of memory
An attempt to allocate more memory failed.

F 409: Illegal object, offset *offset*
Inconsistency found in the object module.

F 410: Illegal object
Inconsistency found in the object module at unknown offset.

F 413: Only *name* object can be linked
It is not possible to link object for other processors.

F 414: Input file *file* same as output file
Input file and output file cannot be the same.

F 415: Demonstration package limits exceeded
One of the limits in this demo version was exceeded.

VERBOSE (V)

V 000: Abort !
The program was aborted by the user.

V 001: Extracting files
Verbose message extracting file from library.

V 002: File currently in progress:
Verbose message file currently processed.

V 003: Starting pass *number*
Verbose message, start of given pass.

V 004: Rescanning....
Verbose message rescanning library. Rescanning is done if there were new unsatisfied externals during the last scan.

V 005: Removing file *file*
Verbose message cleaning up. Temp files are always removed, map file and .out file are removed if switch **-e** is on and the exit code is unequal to zero.

V 006: Object file *file* format *format*
Named object file does not have the standard tool chain object format TIOF-695.

V 007: Library *file* format *format*
Named library file does not have the standard tool chain **ar88** format.

V 008: Embedded environment *name* read, relaxed addressing mode check enabled
Embedded environment successfully read.

APPENDIX D LOCATOR ERROR MESSAGES

Error and warning messages of the locator start with a letter followed by a number and an informational text. The error letter indicates the error type:

W warning
E error
F fatal error
V verbose message

WARNINGS (W)

W 100: Maximum buffer size for *name* is *size* (Adjusted)
For the given format, a maximum buffer size is defined.

W 101: Cannot create map file *filename*, turned off -M option
The given file could not be created.

W 102: Only one -g switch allowed, ignored -g before *name*
Only one .out file can be debugged.

W 104: Found a negative length for section *name*, made it positive
Only stack sections can have a negative length.

W 107: Inserted '*name*' keyword at line *line*
A missing keyword in the description file was inserted.

W 108: Object name (*name*) differs from filename
Internal name of object file not the same as the filename. Maybe renamed?

W 110: Redefinition of system start point
Usually only one load module will access the system table (`__lc_pm`).

W 111: Two -o options, output name will be *name*
Second **-o** option, the message gives the effective name.

W 112: Copy table not referenced, initial data is not copied
If you use a copy statement in the layout part, the initial data is located in rom. Your start-up code should copy this data to their ram location.

W 113: No .out files found to locate
No files where specified at the invocation.

W 114: Cannot find start label *label*
No start point found.

W 116: Redefinition of *name* at line *line*
Identifier was defined twice.

W 119: File *filename* not found in the argument list
All files to be located must be given as an argument.

W 120: unrecognized name option <*name*> at line *line* (inserted '*name*')
Wrong option assignment. Check the manual for possibilities.

W 121: Ignored illegal sub-option '*name*' for *name*
An illegal format sub option was detected. See the format description for this format in the manual.

W 122: Illegal option: *option* (-H or -\? for help)
An illegal option was detected.

W 123: Inserted *character* at line *line*
The given character was missing in the description file.

W 124: Attribute *attribute* at line *line* unknown
An unknown attribute was specified in the description file.

W 125: Copy table not referenced, blank sections are not cleared
Sections with attribute blank are detected, but the copy table is not referenced. The locator generates info for the startup module in the copy table for clearing blank sections at startup. See `__lc_cp` in the manual.

W 127: Layout *name* not found
The used layout in the named file must be defined in the layout part.

W 130: Physical block *name* assigned for the second time to a layout
It is not possible to assign a block more than once to a layout block.

W 136: Removed *character* at line *line*
The character is not needed here.

W 137: Cluster *name* declared twice (layout part)
The named cluster is declared twice. Duplicate cluster names are allowed in the layout part under conditions, because the clusters are referred only. In the layout part the cluster is declared, which may be done only once.

W 138: Absolute section *name* at non-existing memory address 0x*hexnumber*
Absolute section with an address outside physical memory. Either the address is not correct, or the memory description for your target is not consistent.

W 139: *message*
Warning message from the embedded environment. For an overview of the embedded environment error messages, see Appendix F, "Embedded Environment Error Messages".

W 140: File *filename* not found as a parameter
All processes defined in the locator description file (software part) must be specified on the invocation line.

W 141: Unknown space <*name*> in -S option
An unknown space name was specified with a **-S** option.

W 142: No room for section *name* in read-only memory, trying writable memory ...
A section with attribute read-only could not be placed in read-only memory, the section will be placed in writable memory.

ERRORS (E)

E 200: Absolute address 0x*hexnumber* occupied
An absolute address was requested, but the address was already occupied by another section.

E 201: No physical memory available for section *name*
An absolute address was requested, but there is no physical memory at this address.

E 202: Section *name* with mau size *size* cannot be located in an addressing mode with mau size *size*
A bit section cannot be located in a byte oriented addressing mode.

E 203: Illegal object, assignment of non existing var *var*
The MUFOM variable did not exist. For some variables this is an error.

E 204: Cannot duplicate section '*name*' due to hardware limitations
The process must be located more than once, but the section is mapped to a virtual space without memory management possibilities.

E 205: Cannot find section for *name*
Found a variable without a section, should not be possible.

E 206: Size limit for the section group containing section *name* exceeded by 0x*hexnumber* bytes
Small sections do not fit in a page any more.

E 207: Cannot open *filename*
A given file was not found.

E 208: Cannot find a cluster for section *name*
No writable memory available, or unknown addressing mode. Often this error occurs due to an error in the description file.

E 210: Unrecognized keyword <*name*> at line *line*
An unknown keyword was used in the description file.

E 211: Cannot find 0x*hexnumber* bytes for section *name* (fixed mapping)
One of virtual or physical memory was occupied, or there was no physical memory at all!

E 213: The physical memory of *name* cannot be addressed in space *name*
A mapping failed. There was no virtual address space left.

E 214: Cannot map section *name*, virtual memory address occupied
An absolute mapping failed. The memory on the virtual target address was already occupied.

E 215: Available space within *name* exceeded by *number* bytes for section *name*
The available addressing space for an addressing mode has been exceeded.

E 217: No room for section *name* in cluster *name*
The size of the cluster as defined in the .dsc file is too small.

E 218: Missing *identifier* at line *line*
This identifier must be specified.

E 219: Missing ')' at line *line*
Matching bracket missing.

E 220: Symbol '*symbol*' already defined in <*name*>
A symbol was defined twice.

E 221: Illegal object, multi assignment on *var*
The MUFOM variable was assigned more than once, probably due to an error of the object producer.

E 223: No software description found
Each input file must be described in the software description in the .dsc file.

E 224: Missing <length> keyword in block '*name*' at line *line*
No length definition found in hardware description.

E 225: Missing <*keyword*> keyword in space '*name*' at line *line*
For the given mapping, the keyword must be specified.

E 227: Missing <start> keyword in block '*name*' at line *line*
No start definition found in hardware description.

E 230: Cannot locate section *name*, requested address occupied
An absolute address was requested, but the address was already occupied by another process or section.

E 232: Found file *filename* not defined in the description file
All files to be located need a definition record in the description file.

E 233: Environment variable too long in line *line*
Found environment variable in the dsc file contains too many characters.

E 235: Unknown section size for section *name*
No section size found in this .out file. In fact a corrupted .out file.

E 236: Unrecoverable specification at line *line*
An unrecoverable error was made in the description file.

E 238: Found unresolved external(s):
At locate time all externals should be satisfied.

E 239: Absolute address *addr.addr* not found
In the given space the absolute address was not found.

E 240: Virtual memory space *name* not found
In the description files software part for the given file, a non existing memory space was mentioned.

E 241: Object for different processor characteristics
Bits per MAU, MAU per address or endian for this object differs with the first linked object.

E 242: *message*
Error generated by the object. These errors are in fact generated by the assembler. It has been caused by a jump instruction which is out of range.

E 244: Missing *name* part
The given part was not found in the description file, possibly due to a previous error.

E 245: Illegal *name* value at line *line*
A non valid value was found in the description file.

E 246: Identifier cannot be a number at line *line*
A non valid identifier was found in the description file.

E 247: Incomplete type specification, type index = T*hexnumber*
An unknown type was referenced by the given file. Corrupted object file.

E 250: Address conflict between block *block1* and *block2* (memory part)
Overlapping addresses in the memory part of the description file.

E 251: Cannot find 0x*hexnumber* bytes for section *section* in block *block*
No room in the physical block in which the section must be located.

E 255: Section '*name*' defined more than once at line *line*
Sections cannot be declared more than once in one layout/loadmod part.

E 258: Cannot allocate reserved space for process *number*
The memory for a reserved piece of space was occupied.

E 261: User assert: *message*
User-programmed assertion failed. These assertions can be programmed in the layout part of the description file.

E 262: Label '*name*' defined more than once in the software part
Labels defined in the description file must be unique.

E 264: *message*
Error from the embedded environment. For an overview of the embedded environment error messages, see Appendix F, "Embedded Environment Error Messages".

E 265: Unknown section address for absolute section *name*
No section address found in this .out file. In fact a corrupted .out file.

E 266: %s %s not (yet) supported
The requested functionality is not (yet) supported in this release.

FATAL ERRORS (F)

F 400: Cannot create file *filename*
The given file could not be created.

F 401: Cannot open *filename*
A given file was not found.

F 402: Illegal object: Unknown command at offset *offset*
An unknown command was detected in the object file. Corrupted object file.

F 403: Illegal filename (*name*) detected
A filename with an illegal extension was detected on the command line.

F 404: Illegal object: Corrupted hex number at offset *offset*
Wrong byte count in hex number. Corrupted object file.

F 405: Illegal section index
A section index out of range was detected. This could be a corrupted object file, but also a previous error like E 231 (Missing section) is responsible for this message.

F 406: Illegal object: Unknown hex value at offset *offset*
An unknown variable was detected in the object file. Corrupted object file.

F 407: No description file found
The locator must have a description file with the description of the hardware and the software of your system.

F 408: *message*
No protection key or not an IBM compatible PC.

F 410: Only one description file allowed
The locator accepts only one description file.

F 411: Out of memory
An attempt to allocate more memory failed.

F 412: Illegal object, offset *offset*
Inconsistency found in the object module.

F 413: Illegal object
Inconsistency found in the object module at unknown offset.

F 415: Only *name* .out files can be located
It is not possible to locate object for other processors.

F 416: Unrecoverable error at line *line*, *name*
An unrecoverable error was made in the description file in the given part.

F 417: Overlaying not yet done
Overlaying is not yet done for this .out file, link it first without **-r** flag!

F 418: No layout found, or layout not consistent
If there are syntax errors in the layout, it may occur that the layout is not usable for the locator. Syntax errors in the description file must be resolved!

F 419: *message*
Fatal from the embedded environment. For an overview of the embedded environment error messages, see Appendix F, "Embedded Environment Error Messages".

F 420: Demonstration package limits exceeded
One of the limits in this demo version was exceeded.

VERBOSE (V)

V 000: File currently in progress:
Verbose message. On the next lines single filenames are printed as they are processed.

V 001: Output format: *name*
Verbose message for the generated output format.

V 002: Starting pass *number*
Verbose message, start of given pass.

V 003: Abort !
The program was aborted by the user.

V 004: Warning level *number*
Verbose message, report the used warning level.

V 005: Removing file *file*
Verbose message cleaning up. Temporary files are always removed, map file and .out file are removed if switch **-e** is on and the exit code is unequal zero.

V 006: Found file <*filename*> via path *pathname*
The description (include) file was not found in the standard directory. The locator searches also in the install directory etc, in which the file was found.

V 007: *message*
Verbose message from the embedded environment. For an overview of the embedded environment error messages, see Appendix F, "Embedded Environment Error Messages".

APPENDIX E ARCHIVER ERROR MESSAGES

This appendix contains all warnings (W), errors (E) and fatal errors (F) of the archiver **ar88**.

WARNINGS (W)

W 100: Illegal warning level: *level*
Warning level is a single digit.

W 101: Member *name* not found
Library member not found, warning only.

W 102: Can't modify modification time for *name*
The archiver cannot access the file *name* to change the modification time.

W 103: creating archive *name*
The **q** option was used while archive file did not exist (**r** option would be more appropriate).

W 104: Option -a or -b only allowed with key option 'r' or 'm'. Ignored!
Option **a** or **b**, which specifies a position in the archive can only be applied with replace or move actions.

W 105: Only one position specification allowed, ignored '-a or -b *file_offset*'
It is not possible to specify more than one position in the archive. The options **-a** and **-b** are both used to specify a position.

W 106: Option -o only allowed with key option 'x'. Ignored!
Library date can only be preserved with extraction of a library member.

W 107: Option -u only allowed with key option 'r'. Ignored!
Objects newer than the archive are only replaced with key option **r**.

W 108: Option -z only allowed with key option 'r'. Ignored!
Only objects which are moved to the archive can be checked.

W 109: Option -v has no meaning with key option 'p' or 't'. Ignored!
For options **p** and **t** the verbose switch is meaningless.

ERRORS (E)

E 200: filename too long
The filename was too long to fit into the internal buffer.

E 201: Member *name* not found
Library member not found.

E 204: Can't obtain file-status information *filename*
Cannot access *filename* to obtain file status information.

E 207: illegal option: *option*
An illegal option was detected.

FATAL ERRORS (F)

F 300: user abort
The library manager is aborted by the user.

F 301: too much errors
The maximum number of errors is exceeded.

F 302: protection error: *error*
Error message received from ky_init.

F 303: can't create "*filename*"
Cannot create the file with the mentioned name.

F 304: can't open "*filename*"
Cannot open the file with the mentioned name.

F 305: can't reopen '*filename*'
The file *filename* could not be reopened.

F 306: read error while reading "*filename*"
A read error occurred while reading named file.

F 307: write error
A write error occurred while writing to the output file. This error also occurs under DOS when using **-p** and printing the (binary) output to the screen.

F 308: out of memory
An attempt to allocate memory failed.

F 309: illegal character
A character which is not allowed was found.

F 310: *filename* not in archive format
The archive file given is not in the proper format.

F 311: specification of more than one key {rxdmpt} is not permitted
More than one key was given.

F 312: no one of the keys {rxdmpt} was specified
No key was given.

F 313: error in the invocation. Use option -? or -H to get help.
Show usage. For more help, use option **-?**.

F 314: *name* does not exist
Library will only be created in case the **r** key-option is specified.

F 315: IEEE violation for object module *name* at address *address*
IEEE violation detected (**z** option enabled).

F 316: corrupted object module *name*
The object module name does not conform to the IEEE object specification.

F 317: *name*: illegal byte count in hex number, offset = *offset*
Illegal byte count in hex number (IEEE violation).

F 318: evaluation date expired !!

APPENDIX F EMBEDDED ENVIRONMENT ERROR MESSAGES

Error and warning messages from the embedded environment are part of the linker and/or locator error messages. The error numbers mentioned below are not part of the message.

E error
W warning

ERRORS (E)

E 1: Conflicting attributes *attributes* at line *number*
Conflicting attributes.

E 2: Unknown attribute '*character*' at line *number*
Unknown attribute.

E 3: Unknown keyword '*name*' at line *number*
Unknown keyword.

E 4: Illegal character '*character*' at line *number*
Illegal character.

E 5: Page size only allowed in a space definition at line *number*
Page size only allowed in space definition.

E 6: Page size must be a power of 2 at line *number*
Page size must be a power of 2.

E 7: Mau size must be a power of 2 at line *name*
Mau size must be a power of 2.

E 8: Cannot synchronize any more line *number*
Cannot synchronize any more.

E 9: Illegal value '*value*' at line *number*
Illegal value.

E 10: Illegal hex value '*value*' at line *number*
Illegal hex value.

E 11: Illegal octal value '*value*' at line *number*
Illegal octal value.

E 12: Missing value at line *number*
Missing value.

E 13: Illegal identifier at line *number*
Illegal identifier.

E 14: Wrong attribute '*attribute*' at line *number*
Attribute not allowed.

E 15: Unknown identifier '*name*' at line *number*
Unknown identifier.

E 16: Inserted '*character*' at line *number*
Inserted character.

E 17: Cannot find bus/space '*name*' in definition for space '*name*'
Error in the destination of mapping from space.

E 18: Cannot find space/amode '*name*' in definition for amode '*name*'
Map error.

E 19: Cannot find chip '*name*' in definition for bus '*name*'
Map error.

E 20: Cannot find space/amode '*name*' in layout definition for segment '*name*'
Map error.

E 21: Cannot find bus '*name*' in definition for mapping '*name*'
Map error.

WARNINGS (W)

W 100: Cannot find mapping '*name*' in segment definition for space '*name*'
Warning in segment mapping.

APPENDIX G DELFEE

This appendix describes the Delfee description language.

General

description
    partition
    description partition

partition
    memory_partition
    cpu_partition
    software_partition

ident_list
    ident_list , *identifier*
    identifier

identifier
    STRING

file_name
    STRING

CPU

cpu_partition
    cpu { *static_specs_list* }
    cpu { }
    cpu *file_name*

Memory

memory_partition
    memory { *static_specs_list* }
    memory { }
    memory *file_name*

static_specs_list
    static_specs_list static_specs
    static_specs

static_specs
    amod_specs
    spce_specs
    bus_specs
    chips_specs

amod_specs
    amode *ident_list* { *amod_list* }

spce_specs
    space *ident_list* { *spce_list* }

bus_specs
    bus *ident_list* { *bus_list* }

chips_specs
    chips *ident_list chips_list* ;

amod_list
    amod_list amod_def
    amod_def

spce_list
    spce_list spce_def
    spce_def

bus_list
    bus_list bus_def
    bus_def

chips_list
    chips_list chips_def
    chips_def

amod_def
    mau_spec
    attribute_spec
    map_spec

spce_def
    mau_spec
    map_spec

bus_def
    mau_spec
    mem_spec
    map_spec

chips_def
    mau_equ_spec
    attribute_equ_spec
    size_spec

mau_spec
    mau *NUMBER* ;

mau_equ_spec
    mau = *NUMBER*

attribute_spec
    attribute *STRING* ;
    attribute *NUMBER* ;
    attr *STRING* ;
    attr *NUMBER* ;

attribute_equ_spec
    attribute = *STRING*
    attribute = *NUMBER*
    attr = *STRING*
    attr = *NUMBER*

map_spec
    map *map_list* ;

map_list
    map_list map_def
    map_def

map_def
    src_spec
    size_spec
    dst_spec
    align_spec
    page_spec
    amode_spec
    space_spec
    bus_spec

mem_spec
    mem *mem_list* **;**

mem_list
    mem_list mem_def
    mem_def

mem_def
    addr_spec
    chips_spec

src_spec
    src = *NUMBER*

size_spec
    size = *NUMBER*

dst_spec
    dst = *NUMBER*

align_spec
    align = *NUMBER*

page_spec
    page = *NUMBER*

amode_spec
    amode = *identifier*

space_spec
    space = *identifier*

bus_spec
    bus = *identifier*

addr_spec
    address = *NUMBER*
    addr = *NUMBER*

chips_spec
    chips = *low_chip_list*

low_chip_list
    low_chip_list , low_chip_pair
    low_chip_pair

low_chip_pair
    low_chip_pair | low_chip
    low_chip

low_chip
    identifier

Software

software_partition
    software { *layout_blocks* **}**
    software { }
    software *file_name*

layout_blocks
    layout_blocks layout_block
    layout_block

layout_block
    layout
    loadmod

loadmod
    load_mod *software_specs* **;**
    load_mod *identifier software_specs* **;**

software_specs
    software_specs software_spec
    software_spec

software_spec
    start
    process

start
    start = *identifier* **;**

process
    process = *pids*

pids
    NUMBER
    pids , NUMBER

layout
    layout { *space_blocks* **}**
    layout { }
    layout *file_name*

space_blocks
    space_blocks space_block
    space_block

space_block
    space *identifier* **{** *block_blocks* **}**

block_blocks
    block_blocks block_block
    block_block

block_block
block *identifier* **{** *cluster_blocks* **}**

cluster_blocks
cluster_blocks cluster_block
cluster_block

cluster_block
cluster_spec
p_gap_spec
p_fixed_spec
p_pool_spec
p_skip_spec
p_label_spec

cluster_spec
cluster *identifier* **{** *amod_blocks* **}**
cluster *ident_list* **;**

amode_blocks
amode_blocks amode_block
amode_block

amode_block
amode *ident_list* **{** *section_blocks* **}**
amode *ident_list* **;**
section_block

p_gap_spec
gap *length* **;**
gap **;**

p_fixed_spec
fixed *address* **;**

p_pool_spec
pool *length* **;**
pool **;**

p_label_spec
label *identifier* **;**

p_skip_spec
skip **;**

attribute
attribute_equ_spec

length
length = *NUMBER*
leng = *NUMBER*

address
address = *NUMBER*
addr = *NUMBER*

section_blocks
section_blocks section_block
section_block

section_block
section_spec
copy_spec
v_fixed_spec
v_gap_spec
v_reserved_spec
stack_spec
heap_spec
table_spec
others
v_label_spec
v_assert_spec
attribute_spec

section_spec
section *selection modifiers* **;**
section *selection* **;**

modifiers
modifiers modifier
modifier

modifier
attribute
address

copy_spec
copy *selection attribute* **;**
copy *selection* **;**
copy **;**

selection
selection = *STRING*
identifier

v_fixed_spec
fixed *address* **;**

v_gap_spec
gap **;**

v_reserved_spec
reserved *reserved_options* **;**
reserved **;**

reserved_options
reserved_options reserved_option
reserved_option

reserved_option
attribute
address
length
v_label_equ_spec

stack_spec
stack *stack_options* **;**
stack **;**

heap_spec
    heap *stack_options* **;**
    heap ;

stack_options
    stack_options stack_option
    stack_option

stack_option
    attribute
    length

table_spec
    table *attribute* **;**
    table ;

v_label_spec
    label *identifier* **;**

v_label_equ_spec
    label = *identifier*

v_assert_spec
    assert (*bool_expression* **,** *STRING* **) ;**
    asse (*bool_expression* **,** *STRING* **) ;**

others
    others ;

bool_expression
    termp bool_op termp

termp
    term **+** *termp*
    term **-** *termp*
    term

term
    (*term* **)**
    identifier
    NUMBER

bool_op
    <
    >
    ==
    !=

A **NUMBER** is a series of (hex) digits with optional suffixes 'k' 'M' 'G' which stands for 'kilo', 'mega' and 'giga'. Numbers may be given in hex, octal or decimal with the usual prefix. Where applicable numbers may be preceded by a minus sign.

A **STRING** is a series of characters that is not a number (089 is a **STRING** because it is not a valid octal number) and consists of alphanumeric characters including '_', '.', '-' and the directory separators. ('\', '/' and ':')

Any (part of a) token may contain environment variables. If the environment variable A contains the text 'foo' then the sequence:

```
$A/proto.dsc
```

is translated to:

```
foo/proto.dsc
```

Multi character variables must be combined with braces:

```
window = $(MODE);
```

There are three methods to write comments in a delfee script. The first one is the 'C' style comment between '/*' and '*/'. The second form is a '#' in the first column. The second form allows preprocessing by the C-preprocessor. Any *#line* or *#file* directive will be ignored by the locator. The third form is the 'C++' style comment; a double slash '//' anywhere on a line introduces comments until the end of line.

APPENDIX H IEEE-695 OBJECT FORMAT

H.1 IEEE-695

The IEEE-695 standard describes MUFOM: Microprocessor Universal Format for Object Modules. It defines a target independent storage standard for object files. However, this standard does not describe how symbolic debug information should be encoded according to that standard. Symbolic debug information can be a part of an object file. A debugger which reads an object file uses the symbolic debug information to obtain knowledge about the relation between the executable code and the origination high-level language source files. Since the IEEE-695 standard does not describe the representation of debug information, working implementations of this standard show vendor specific and microprocessor specific solutions for this area.

H.2 Command Language Concept

Most object formats are record oriented: there are one or more section headers at a fixed position in the file which describe how many sections are present. A section header contains information like start address, file offset, etc. The contents of the section is in some data part, which can only be processed after the header has been read. So the tool that reads such an object uses implicit assumptions how to process such a file. Seeking through the file to get those records which are relevant is usual.

MUFOM (IEEE-695) uses a different approach. It is designed as a command language which steers the linker, locator and object reader in the debugger.

An assembler or compiler may create an object module where most of the data contained in it is relocatable. The next phase in the translation process is linking several object modules into one new object module. A relocatable object uses relocation expressions at places where the absolute values are not yet known. An expression evaluator in the locator transforms the relocation expressions into absolute values.

Finally the object is ready for loading into memory. Since an object file is transformed by several processes, MUFOM implements an object file as a sequence of commands which steers this transformation process.

These commands are created, executed or copied by one of five processes which act on a MUFOM object file:

1. Creation process
   Creation of the object file by an assembler or compiler. The assembler or compiler tells other MUFOM processes what to do, by emitting commands generated from assembly source text or a high-level language.
2. Linkage process
   Linking of several object modules into one module resolving external references by renaming X variables into I variables, and by generating new commands (assigning of R variables).
3. Relocation process
   Relocation, giving all sections an absolute address by assigning their L variable.
4. Expression evaluation process
   Evaluation of loader expressions, generated in one of the three previously mentioned MUFOM processes.
5. Loader process
   Loading the absolute memory image.

The last four processes are in fact command interpreters: the assembler writes an object file which is basically a large sequence of instructions for the linker. For example, instead of writing the contents of a section as a sequence of bytes at a specific position in the file, IEEE-695 defines a load command, LR, which instructs the linker to load a number of bytes. The LR command specifies the number of MAUs (minimum addressable unit) that will be relocated, followed by the actual data. This data can be a number of absolute bytes, or an expression which must be evaluated by the linker.

Transforming relocation expressions into new expressions or absolute data and combining sections is the actual linkage process.

It is possible that one or more of the above MUFOM processes are combined in one tool. For instance, the locator is built from process 3 and process 4 above.

H.3 Notational Conventions

The following conventions are used in this appendix:

|	select one of the items listed between '|'
" "	literal characters are between " "
[ ]+	optional item repeats one time or more
[ ]?	optional item repeats zero times or one time
[ ]*	optional item repeats zero times or more
::=	can be read as "is defined as"

H.4 Expressions

An expression in an IEEE-695 file is a combination of variables, operators and absolute data.

The variable name always starts with a non-hexadecimal letter (G...Z), immediately followed by an optional hexadecimal number. The first non-hexadecimal letter gives the class of the variable. Reading an object file you encounter the following variables:

G - Start address of a program. If not assigned this address defaults to the address of low-level symbol **_start**.

I - An I variable represents a global symbol in an object module.
The I variable is assigned an expression which is to be made available to other modules for the purpose of linkage edition. The name of an I variable is always composed of the letter 'I', followed by a hexadecimal number. An I variable is created only by an NI command.

L - Start address of a section. This variable is only used for absolute sections. The 'L' is followed by a section index, which is an hexadecimal number. L variables are created by an assignment command, but the section index must have been defined by an ST command.

N - Name of internal symbol. This variable is used to assign values of local symbols, or, to build complex types for use by a high-level language debugger, or for inter-modular type checking during linkage. The N variable is created with a NN command.

P - Program pointer per section. This variable always contains the current address of the target memory location. The P variable is followed by a section index, which is a hexadecimal number. The section index must have been defined with an ST command (section type command). The variable is created after its first assignment.

R - The R type variable is a relocation reference for a particular section. All references to addresses in this section must be made relative to the R variable. Linking is accomplished by assigning a new value to R. The R variable consists of the letter 'R', followed by a section index, which is a hexadecimal number. The section index must have been defined with an ST command. The default value of an (unassigned) R variable is 0.

S - The S type variable is the section size (in MAUs) for a section. There is one S variable per section. The 'S' is followed by an section index. An S variable is created by its first assignment.

W - Work variable. This type of variable can be used to assign values to, which can be used in following MUFOM commands. They serve the purpose of maintaining values in a workspace without any additional meaning. A work variable consists of the letter 'W' followed by a hexadecimal number. W variables are created by their first assignment.

X - An X type variable refers to an external reference. X-variables cannot have a value assigned to it. An X variable consists of the letter 'X' followed by a hexadecimal number.

The MUFOM language uses the following data types to form expressions:

```
digit            ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
hex_letter       ::= "A" | "B" | "C" | "D" | "E" | "F"
hex_digit        ::= digit | hex_letter
hex_number       ::= [ hex_digit ]+
nonhex_letter    ::= "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" |
                     "U" | "V" | "W" | "X" | "Y" | "Z"
letter           ::= hex_letter | nonhex_letter
alpha_num        ::= letter | digit
identifier       ::= letter [ alpha_num ]*
character        ::= 'value valid within chosen character set'
char_string_length ::= hex_digit hex_digit
char_string      ::= char_string_length [ character ]*
```

The numeric value specified in 'char_string_length' should be followed by an equal number of characters.

Expressions may be formed out of immediate numbers and MUFOM variables. The MUFOM processes 2 to 4, which form the linker and the locator, contain expression evaluators which parse and calculate the values for the expressions. If a MUFOM process cannot calculate the absolute value of an expression, because the values of the variable are not yet known, it copies the expression (with modifications) into the output file.

Expression are coded in reverse Polish notation. (The operator follows the operands.)

expression ::= boolean_function | one_operand_function | two_operand_function | three_operand_function | four_operand_function | conditional_expr | hex_number | MUFOM_variable

H.4.1 Functions without Operands

@F : false function
@T : true function

boolean_function ::= "@F" | "@T"

The false and true function produce a boolean result false or true which may be used in logical expressions. Both functions do not have operands.

H.4.2 Monadic Functions

Monadic functions have one operand which precedes the function.

one_operand_function ::= operand "," monop
operand ::= expression
monop ::= "@ABS" | "@NEG" | "@NOT" | "@ISDEF"

@ABS : returns the absolute value of an integer operand.

@NEG : returns the negative value of an integer operand.

@NOT : returns the negation of a boolean operand or the one's complement value if the operand is an integer

@ISDEF : returns the logical true value if all variable in an expression are defined, return false otherwise.

H.4.3 Dyadic Functions and Operators

Dyadic functions and operators have two operands which precede the operator or function.

two_operand_function ::= operand1 "," operand2 "," dyadop
operand1 ::= expression
operand2 ::= expression
dyadop ::= "@AND" | "@MAX" | "@MIN" | "@MOD" | "@OR" |"@XOR" | "+" | "-" | "/" | "*" | "<" | ">" | "=" | "#"

@AND : returns boolean true/false result of logical 'and' operation on operands, when both operands are logical values. When both operands are not logical values the bitwise and is performed.

@MAX : compares both operands arithmetically and returns the largest value.

@MIN : compares both operands arithmetically and returns the smallest value.

@MOD : returns the modulo result of the division of operand1 by operand2. The result is undefined if either operand is negative, or if operand2 is zero.

@OR : returns boolean true/false result of logical 'or' operation on operands, when both operands are logical values. When both operands are no logical values the bitwise and is performed.

+, -, *, / : These are the arithmetic operators for addition, subtraction, multiplication and division. The result is an integer. For division the result is undefined if operand2 equals zero. The result of a division rounds toward zero.

<, >, =, # : These are operators for the following logical relations: 'less than', 'greater than', 'equals', 'is unequal'. The result is true or false.

H.4.4 MUFOM Variables

The meaning of the MUFOM variable is explained in Section H.4. The following syntax rules apply for the MUFOM variables:

```
MUFOM_variable      ::= MUFOM_var |
                        MUFOM_var_num
                        MUFOM_var_optnum
MUFOM_var           ::= "G"
MUFOM_var_num       ::= "I" | "N" | "W" | "X"
                        hex_number
MUFOM_var_optnum    ::= "L" | "P" | "R" | "S"
                        [ hex_number ]?
```

H.4.5 @INS and @EXT Operator

The @INS operator inserts a bit string.

four_operand_function ::= operand1 "," operand2 "," operand3 "," operand4 "," @INS

operand2 is inserted in operand1 starting at position operand3, and ending at position operand4.

The @EXT operator extracts a bit string.

three_operand_function ::= operand1 "," operand2 "," operand3 "," @EXT

A bit string is extracted from operand1 starting at position operand2 and ending at position operand3.

H.4.6 Conditional Expressions

```
conditional_expr ::= err_expr | if_else_expr
err_expr         ::= value "," condition "," err_num "," "@ERR"
value            ::= expression
condition        ::= expression
err_num          ::= expression
if_else_expr     ::= condition "," "@IF" "," expression "," "@ELSE" "," expression "," "@END"
```

H.5 MUFOM Commands

H.5.1 Module Level Commands

At module level there are four commands: one command to start and one to end a module, one command to set the date and time of creation of the module, and one command to specify address formats.

H.5.1.1 MB Command

The MB command is the first command in a module. It specifies the target machine configuration and an optional command with the module name.

MB_command ::= "MB" machine_identifier ["," module_name]? "."

Example: `MB S1C88.`

H.5.1.2 ME Command

The module end command is the last command in an object file. It defines the end of the object module.

ME_command ::= "ME."

H.5.1.3 DT Command

The DT command sets the date and time of creation of an object module.

DT_command ::= "DT" [digit]* "."

Example: `DT19930120120432.`

The format of display of the date and time is "YYYYMMDDHHMMSS":

4 digits for the year, 2 digits for the month, 2 digits for the day, 2 digits for the hour, 2 digits for the minutes and 2 digits for the seconds.

H.5.1.4 AD Command

The AD command specifies the address format of the target execution environment.

```
AD_command      ::= "AD" bits_per_MAU [ "," MAU_per_address [ "," order ]? ]?
MAU_per_address ::= hex_number
bits_per_MAU    ::= hex_number
order           ::= "L" | "M"
```

MAU stands for minimum addressable unit. This is target processor dependant.

L means least significant byte at lowest address (little endian)
M means most significant byte at lowest address (big endian)

Example: `AD8,3,L.` Specifies a 3-byte addressable 8-bit processor running in little endian mode.

H.5.2 Comment and Checksum Command

The comment command offers the possibility to store information in an object module about the object module and the translators that created it. The comment may be used to record the file name of the source file of the object module or the version number of the translator that created it. Because the standard supports several layers each of which has its own revision number an object module may contain several comment commands which specify which revision of the standard has been used to create the module. The contents of a comment is not prescribed by the standard and thus it is implementation defined how a MUFOM process handles a comment command.

```
CO_command   ::= "CO" [comment_level]? "," comment_text "."
comment_level ::= hex_number
comment_text  ::= char_string
```

The comment levels 0–6 are reserved to pass information about the revision number of the layers in this standard.
The checksum command starts and checks the checksum calculation of an object module.

H.5.3 Sections

A section is the smallest unit of code or data that can be controlled separately. Each section has a unique number which is introduced at the first section begin (SB) command. The contents of a section may follow its introduction. A section ends at the next SB command with a number different from the current number. A section resumes at an SB command with a number that has been introduced before.

H.5.3.1 SB Command

SB_command ::= "SB" hex_number "."

The maximum number of sections in an object module is implementation defined.

H.5.3.2 ST Command

The ST command specifies the type of a section.

ST_command ::= "ST" section_number ["," section_type]* ["," section_name]? "."
section_type ::= letter
section_name ::= char_string

A section can be named or unnamed. If section_name is omitted a section is unnamed. A section can be relocatable or absolute. If the section start address is an absolute number the section is called absolute. If the section start address is not yet known, the section is called relocatable. In relocatable sections all addresses are specified relative to the relocation base of that section. The relocation phase of the linker or locator may map the relocation base of a section onto a fixed address.

During linkage edition the section name and the section attributes identify a section and thus the actions to be taken. If a section is defined in several modules, the linkage editor must determine how to act on sections with the same name. This can be either one of the following strategies:

- several sections are to be joined into a single one
- several sections are to be overlapped
- sections are not to coexist

A section type gives additional information to the linkage editor about the section, which may be used to layout a section in memory. Section type information is encoded with letters, which may be combined in one ST command. Some combinations of letters are invalid or may be meaningless.

Letter	Meaning	Class	Explanation
A	absolute	access	section has absolute address assigned to corresponding L-variable
R	read only	access	no write access to this section
W	writable	access	section may be read and written
X	executable	access	section contains executable code
Z	zero page	access	if target has zero page or short addressable page Z-section map into it
Y*num*	addressing mode	access	section must be located in addressing mode *num*
B	blank	access	section must be initialized to '0' (cleared)
F	not filled	access	section is not filled or cleared (scratch)
I	initialize	access	section must be initialized in rom
E	equal	overlap	if sections in two modules have different length an error must be raised
M	max	overlap	Use largest value as section size
U	unique	overlap	The section name must be unique
C	cumulative	overlap	Concatenate sections if they appear in several modules. The section alignment for partial section must be preserved
O	overlay	overlap	sections with the name *name@func* must be combined to one section *name*, according to the rules for *func* obtained from the call graph
S	separate	overlap	multiple sections can have the same name and they may relocated at unrelated addresses
N	now	when	section is located before normal sections (without N or P)
P	postpone	when	section is located after normal sections (without N or P)

H.5.3.3 SA Command

```
SA_command     ::= "SA" section_number "," [ MAU_boundary ]? [ "," page_size ]? "."
MAU_boundary   ::= expression
page_size      ::= expression
```

The MAU boundary value forces the relocator to align a section on the number of MAUs specified. If page_size is present the relocator checks that the section does not exceed a page boundary limit when it is relocated.

H.5.4 Symbolic Name Declaration and Type Definition

H.5.4.1 NI Command

The NI command defines an internal symbol. An internal symbol is visible outside the module. Thus it may resolve an undefined external in another module.

```
NI_command ::= "N" I_variable "," char_string "."
```

The NI_command must precede any reference to the I_variable in a module. There may not be more than one I_variable with the same name or number.

H.5.4.2 NX Command

The NX command defines an external symbol which is undefined in the current module. The NX command must precede all occurrences of the corresponding X variable.

```
NX_command ::= "N" X_variable "," char_string "."
```

The unresolved reference corresponding to an NX-command can be resolved by an internal symbol definition (NI_command) in another module.

H.5.4.3 NN Command

The NN command defines a local name which may be used for defining a name of a local symbol in a module or a name in a type definition.

A name defined with an NN command is not visible outside the scope of the module. The NN command must precede all occurrences of the corresponding N variable.

```
NN_command ::= "N" N_variable "," char_string "."
```

H.5.4.4 AT Command

The attribute command may be used to define debugging related information of a symbol, such as the symbol type number. Level 2 of the standard does not prescribe the contents of the optional fields of the AT command. The language dependent layer (level 3) describes how these fields can be used to pass high-level symbol information with the AT command.

```
AT_command     ::= "AT" variable "," type_table_entry [ "," lex_level [ "," hex_number ]* ]? "."
variable       ::= I_variable | N_variable | X_variable
type_table_entry ::= hex_number
lex_level      ::= hex_number
```

The type_table entry is a type number introduced with a type command (TY). References to type numbers in the AT command may precede the definition of the type in the TY command.

The meaning of the lex_level field is defined at layer 3 or higher. The same applies to the optional hex_number fields.

H.5.4.5 TY Command

The TY-command defines a new type table entry. The type number introduced by the type command can be seen as a reference index to this type. The TY-command defines the relation between the newly introduced type and other types that are defined in other places in the object module. It also establishes a relation between a new type index and symbols (N_variable).

```
TY_command      ::= "TY" type_table_entry [ "," parameter ]+ "."
type_table_entry ::= hex_number
parameter       ::= hex_number | N_variable | "T" type_table_entry
```

Level 2 does not define the semantics of the parameters. These are defined at level 3, the language layer. A linkage editor which does not have knowledge of the semantics of the parameter in a type command can still perform type comparison: Two types are considered to compare equal when the following conditions hold:

- both types have an equal number of parameters
- the numeric values in the types are equal
- N_variables in both types have the same name
- the type entries referenced from both types compare equal

Variable N0 is supposed to compare equal to any other name.

Type table entry T0 is supposed to compare equal to any other type.

H.5.5 Value Assignment

H.5.5.1 AS Command

The assignment command assigns a value to a variable.

```
AS_command ::= "AS" MUFOM_variable "," expression "."
```

H.5.6 Loading Commands

The contents of a section is either absolute data (code) or relocatable data (code). Absolute data can be loaded with the LD command. The address where loading takes place depends on the value of the P-variable belonging to the section. Data which is contiguous in a LD command is supposed to be loaded contiguously in memory.

If data is not absolute it contains expressions which must be evaluated by the expression evaluator. The LR command allows a relocation expression to be part of the loading command.

H.5.6.1 LD Command

```
LD_command ::= "LD" [ hex_digit ]+ "."
```

The constants loaded with the LD command are loaded with the most significant part first.

H.5.6.2 IR Command

A relocation base is an expression which can be associated with a relocation letter. This relocation letter can be used in subsequent load relocate commands.

```
IR_command      ::= "IR" relocation_letter "," relocation_base [ "," number_of_bits ]? "."
relocation_letter ::= nonhex_letter
relocation_base  ::= expression
number_of_bits  ::= expression
```

Example:
```
IRV,X20,16.
ITM,R2,40,+,8.
```

The number_of_bits must be less than or equal to the number of bits per address, which is the product of the number of MAUs per address and the number of bits per MAU, both of which are specified in the AD command. If the number_of_bits is not specified it equals the number of bits per address.

H.5.6.3 LR Command

```
LR_command       ::= "LR" [ load_item ]+ "."
load_item        ::= relocation_letter offset "," | load_constant |
                     "(" expression [ "," number_of_MAUs ]? ")"
load_constant    ::= [ hex_digit ]+
number_of_MAUs ::= expression
```

Example:

```
LR002000400060.
LRT80,0020.
LR(R2,100,+,4).
```

The first example shows immediate constants which may be loaded as a part of an LR command.
The second example shows the use of the relocation base defined in the previous paragraph, followed by a constant.
The third example shows how the value of the expression R2 + 100 is used to load 4 MAUs.
The three commands in this example may be combined into one LR command:

```
LR002000400060T80,0020(R2,100,+,4).
```

H.5.6.4 RE Command

The replicate command defines the number of times a LR command must be replicated:

```
RE_command ::= "RE" expression "."
```

The LR command must immediately follow the RE command.

Example:

```
RE04.
LR(R2,200,+,4).
```

The commands above load 16 MAUs: 4 times the 4 MAU value of R2 + 200.

H.5.7 Linkage Commands

H.5.7.1 RI Command

The retain internal symbol command indicates that the symbolic information of an NI command must be retained in the output file.

```
RI_command  ::= "R" I_variable [ "," level_number ]? "."
level_number ::= hex_number
```

H.5.7.2 WX Command

The weak external command flags a previously defined external (NX_command) as weak. This means that if the external remains unresolved, the value of the expression in the WX command is assigned to the X variable.

```
WX_command ::= "W" X_variable [ "," default_value ]? "."
default_value  ::= expression
```

H.5.7.3 LI Command

The LI command specifies a default library search list. The library names specified in the LI_command are searched for unresolved references.

```
LI_command ::= "LI" char_string [ "," char_string ]* "."
```

H.5.7.4 LX Command

The LX command specifies a library to search for a named unresolved variable.

```
LX_command ::= "L" X_variable [ "," char_string ]+ "."
```

The paragraphs above showed the commands and operators as ASCII strings. In an object file they are binary encoded. The following tables show the binary representation.

H.6 MUFOM Functions

The following table lists the first byte of MUFOM elements. Each value between 0 and 255 classifies the MUFOM language element that follows, or it is a language element itself. E.g. numbers outside the range 0–127 are preceded by a length field: 0x82 specifies that a 2 byte integer follows. 0xE4 is the function code for the LR command.

Overview of first byte of MUFOM language elements

0x00–0x7F	Start of regular string, or one byte numbers ranging from 0–127
0x80	Code for omitted optional number field
0x81–0x88	Numbers outside the range 0–127
0x89–0x8F	Unused
0x90–0xA0	User defined function codes
0xA0–0xBF	MUFOM function codes
0xC0	Unused
0xC1–0xDA	MUFOM letters
0xDB–0xDF	Unused
0xE0–0xF9	MUFOM commands
0xFA–0xFF	Unused

Binary encoding of MUFOM letters and function codes

Function code		Identifiers	
Function	**Code**	**Letter**	**Code**
@F	0xA0		
@T	0xA1	A	0xC1
@ABS	0xA2	B	0xC2
@NEG	0xA3	C	0xC3
@NOT	0xA4	D	0xC4
+	0xA5	E	0xC5
-	0xA6	F	0xC6
/	0xA7	G	0xC7
*	0xA8	H	0xC8
@MAX	0xA9	I	0xC9
@MIN	0xAA	J	0xCA
@MOD	0xAB	K	0xCB
<	0xAC	L	0xCC
>	0xAD	M	0xCD
=	0xAE	N	0xCE
!= <>	0xAF	O	0xCF
@AND	0xB0	P	0xD0
@OR	0xB1	Q	0xD1
@XOR	0xB2	R	0xD2
@EXT	0xB3	S	0xD3
@INS	0xB4	T	OxD4
@ERR	0xB5	U	0xD5
@IF	0xB6	V	0xD6
@ELSE	0xB7	W	0xD7
@END	0xB8	X	0xD8
@ISDEF	0xB9	Y	0xD9
		Z	0xDA

MUFOM command codes

Command	Code	Description
MB	0xE0	Module begin
ME	0xE1	Module end
AS	0xE2	Assign
IR	0xE3	Initialize relocation base
LR	0xE4	Load with relocation
SB	0xE5	Section begin
ST	0xE6	Section type
SA	0xE7	Section alignment
NI	0xE8	Internal name
NX	0xE9	External name
CO	0xEA	Comment
DT	0xEB	Date and time
AD	0xEC	Address description
LD	0xED	Load
CS (with sum)	0xEE	Checksum followed by sum value
CS	0xEF	Checksum (reset sum to 0)
NN	0xF0	Name
AT	0xF1	Attribute
TY	0xF2	Type
RI	0xF3	Retain internal symbol
WX	0xF4	Weak external
LI	0xF5	Library search list
LX	0xF6	Library external
RE	0xF7	Replicate
SC	0xF8	Scope definition
LN	0xF9	Line number
	0xFA	Undefined
	0xFB	Undefined
	0xFC	Undefined
	0xFD	Undefined
	0xFE	Undefined
	0xFF	Undefined

APPENDIX I MOTOROLA S-RECORDS

The locator generates three types of S-records: S0, S2 and S8. They have the following layout:

S0 - record

'S' '0' <*length_byte*> <2 bytes 0> <*comment*> <*checksum_byte*>

A locator generated S-record file starts with a S0 record with the following contents:

length_byte : 10H
comment : E0C88 locator
checksum : 88H

```
        E 0 C 8 8   l o c a t o r
S0100000534D433838206C6F6361746F7288
```

The S0 record is a comment record and does not contain relevant information for program execution.

The length_byte represents the number of bytes in the record, not including the record type and length byte.

The checksum is calculated by first adding the binary representation of the bytes following the record type (starting with the length_byte) to just before the checksum. Then the one's complement is calculated of this sum. The least significant byte of the result is the checksum. The sum of all bytes following the record type is 0FFH.

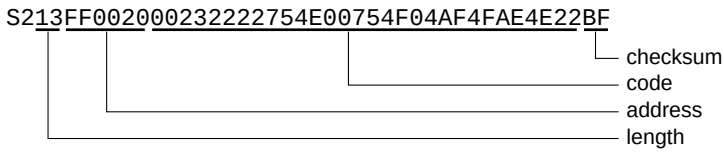
S2 - record

The actual program code and data is supplied with S2 records, with the following layout:

'S' '2' <*length_byte*> <*address*> <*code bytes*> <*checksum_byte*>

For the S1C88 the locator generates 3-byte addresses.

Example:



The length of the output buffer for generating S2 records is 32 code bytes.

The checksum calculation of S2 records is identical to S0.

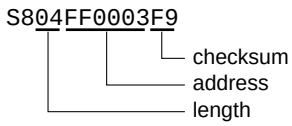
S8 - record

At the end of an S-record file, the locator generates an S8 record, which contains the program start address.

Layout:

'S' '8' <*length_byte*> <*address*> <*checksum_byte*>

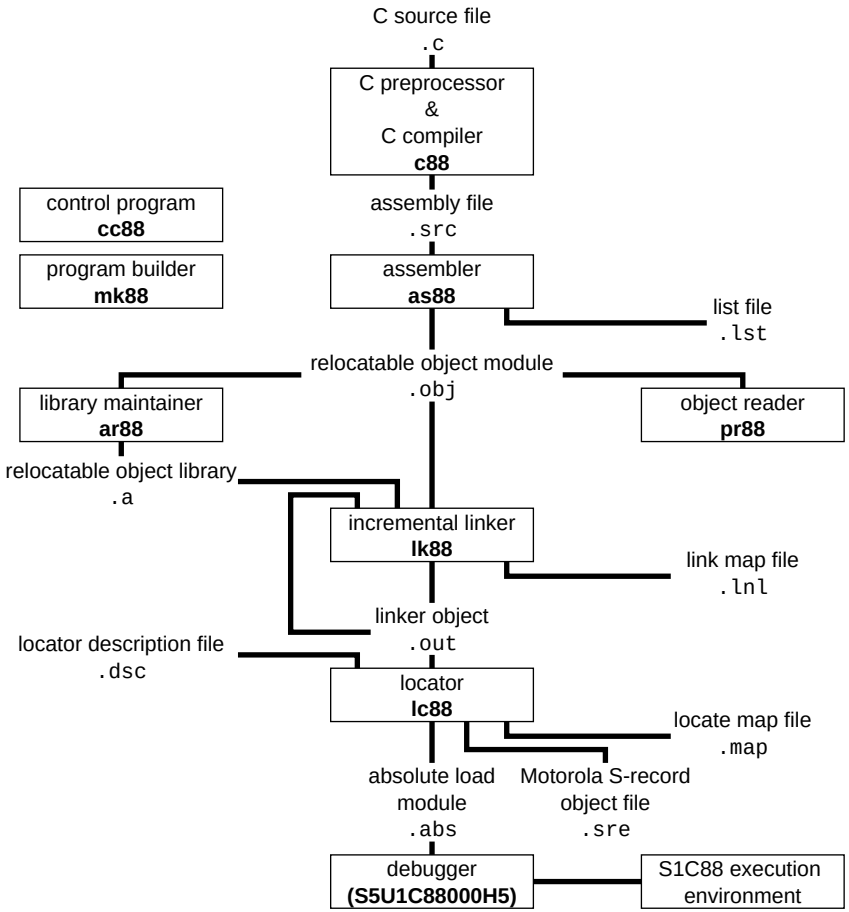
Example:



The checksum calculation of S8 records is identical to S0.

S1C88 Family C Compiler

Quick Reference


C source file
.c
C preprocessor
&
C compiler
c88
control program
cc88
assembly file
.src
program builder
mk88
assembler
as88
list file
.lst
relocatable object module
.obj
library maintainer
ar88
object reader
pr88
relocatable object library
.a
incremental linker
lk88
link map file
.lnl
linker object
.out
locator description file
.dsc
locator
lc88
locate map file
.map
absolute load
module
.abs
Motorola S-record
object file
.sre
debugger
(S5U1C88000H5)
S1C88 execution
environment

Startup Command

c88 [[*option*]...[*file*]...]...

Options

Include options

-f *file*	Read options from *file*
-H *file*	Include *file* before starting compilation
-I*directory*	Look in *directory* for include files

Preprocess options

-D*macro*[=*def*]	Define preprocessor *macro*

Code generation options

-M{**s**|**c**|**d**|**l**}	Select memory model: small, compact code, compact data or large
-O{**0**|**1**}	Control optimization

Output file options

-e	Remove output file if compiler errors occur
-o *file*	Specify name of output *file*
-s	Merge C-source code with assembly output

Diagnostic options

-V	Display version header only
-err	Send diagnostics to error list file (`.err`)
-g	Enable symbolic debug information
-w[*num*]	Suppress one or all warning messages

Error/Warning Messages

I: information E: error F: fatal error S: internal compiler error W: warning

Frontend

F 1:	evaluation expired	Your product evaluation period has expired.
W 2:	unrecognized option: '*option*'	The option you specified does not exist.
E 4:	expected *number* more '#endif'	The preprocessor part of the compiler found the '**#if**', '**#ifdef**' or '**#ifndef**' directive but did not find a corresponding '**#endif**' in the same source file.
E 5:	no source modules	You must specify at least one source file to compile.
F 6:	cannot create "*file*"	The output file or temporary file could not be created.
F 7:	cannot open "*file*"	Check if the file you specified really exists.
F 8:	attempt to overwrite input file "*file*"	The output file must have a different name than the input file.
E 9:	unterminated constant character or string	This error can occur when you specify a string without a closing double-quote (") or when you specify a character constant without a closing single-quote (').
F 11:	file stack overflow	This error occurs if the maximum nesting depth (50) of file inclusion is reached.
F 12:	memory allocation error	All free space has been used.
W 13:	prototype after forward call or old style declaration - ignored	Check that a prototype for each function is present before the actual call.
E 14:	';' inserted	An expression statement needs a semicolon.
E 15:	missing filename after -o option	The **-o** option must be followed by an output filename.
E 16:	bad numerical constant	A constant must conform to its syntax. Also, a constant may not be too large to be represented in the type to which it was assigned.
E 17:	string too long	This error occurs if the maximum string size (1500) is reached.
E 18:	illegal character (0x*hexnumber*)	The character with the hexadecimal ASCII value 0x*hexnumber* is not allowed here.
E 19:	newline character in constant	The newline character can appear in a character constant or string constant only when it is preceded by a backslash (\).
E 20:	empty character constant	A character constant must contain exactly one character. Empty character constants ('') are not allowed.
E 21:	character constant overflow	A character constant must contain exactly one character. Note that an escape sequence is converted to a single character.
E 22:	'#define' without valid identifier	You have to supply an identifier after a '**#define**'.

Error/Warning Messages

Frontend

E 23:	'#else' without '#if'	'**#else**' can only be used within a corresponding '**#if**', '**#ifdef**' or '**#ifndef**' construct.
E 24:	'#endif' without matching '#if'	'**#endif**' appeared without a matching '**#if**', '**#ifdef**' or '**#ifndef**' preprocessor directive.
E 25:	missing or zero line number	'**#line**' requires a non-zero line number specification.
E 26:	undefined control	A control line (line with a '#*identifier*') must contain one of the known preprocessor directives.
W 27:	unexpected text after control	'**#ifdef**' and '**#ifndef**' require only one identifier. Also, '**#else**' and '**#endif**' only have a newline. '**#undef**' requires exactly one identifier.
W 28:	empty program	The source file must contain at least one external definition. A source file with nothing but comments is considered an empty program.
E 29:	bad '#include' syntax	A '**#include**' must be followed by a valid header name syntax.
E 30:	include file "*file*" not found	Be sure you have specified an existing include file after a '**#include**' directive. Make sure you have specified the correct path for the file.
E 31:	end-of-file encountered inside comment	The compiler found the end of a file while scanning a comment. Probably a comment was not terminated.
E 32:	argument mismatch for macro "*name*"	The number of arguments in invocation of a function-like macro must agree with the number of parameters in the definition. Also, invocation of a function-like macro requires a terminating ")" token.
E 33:	"*name*" redefined	The given identifier was defined more than once, or a subsequent declaration differed from a previous one.
W 34:	illegal redefinition of macro "*name*"	A macro can be redefined only if the body of the redefined macro is exactly the same as the body of the originally defined macro.
E 35:	bad filename in '#line'	The string literal of a **#line** (if present) may not be a "**wide-char**" string.
W 36:	'debug' facility not installed	'**#pragma debug**' is only allowed in the debug version of the compiler.
W 37:	attempt to divide by zero	A divide or modulo by zero was found.
E 38:	non integral switch expression	A `switch` condition expression must evaluate to an integral value.
F 39:	unknown error number: *number*	This error may not occur.
W 40:	non-standard escape sequence	Your escape sequence contains an illegal escape character.

Frontend

E 41:	'#elif' without '#if'	The '**#elif**' directive did not appear within an '**#if**', '**#ifdef**' or '**#ifndef**' construct.
E 42:	syntax error, expecting parameter type/declaration/ statement	A syntax error occurred in a parameter list a declaration or a statement.
E 43:	unrecoverable syntax error, skipping to end of file	The compiler found an error from which it could not recover.
I 44:	in initializer "*name*"	Informational message when checking for a proper constant initializer.
E 46:	cannot hold that many operands	The value stack may not exceed 20 operands.
E 47:	missing operator	An operator was expected in the expression.
E 48:	missing right parenthesis	')' was expected.
W 49:	attempt to divide by zero - potential run-time error	An expression with a divide or modulo by zero was found.
E 50:	missing left parenthesis	'(' was expected.
E 51:	cannot hold that many operators	The state stack may not exceed 20 operators.
E 52:	missing operand	An operand was expected.
E 53:	missing identifier after 'defined' operator	An identifier is required in a #if defined(*identifier*).
E 54:	non scalar controlling expression	Iteration conditions and '**if**' conditions must have a scalar type (not a struct, union or a pointer).
E 55:	operand has not integer type	The operand of a '**#if**' directive must evaluate to an integral constant.
W 56:	'<*debugoption*><*level*>' no associated action	There is no associated debug action with the specified debug option and level.
W 58:	invalid warning number: *number*	The warning number you supplied to the **-w** option does not exist.
F 59:	sorry, more than number errors	Compilation stops if there are more than 40 errors.
E 60:	label "*label*" multiple defined	A label can be defined only once in the same function.
E 61:	type clash	The compiler found conflicting types.
E 62:	bad storage class for "*name*"	The storage class specifiers `auto` and `register` may not appear in declaration specifiers of external definitions. Also, the only storage class specifier allowed in a parameter declaration is `register`.
E 63:	"*name*" redeclared	The specified identifier was already declared. The compiler uses the second declaration.

Error/Warning Messages

Frontend

E 64:	incompatible redeclaration of "*name*"	The specified identifier was already declared.
W 66:	function "*name*": variable "*name*" not used	A variable is declared which is never used.
W 67:	illegal suboption: *option*	The suboption is not valid for this option.
W 68:	function "*name*": parameter "*name*" not used	A function parameter is declared which is never used.
E 69:	declaration contains more than one basic type specifier	Type specifiers may not be repeated.
E 70:	'break' outside loop or switch	A `break` statement may only appear in a `switch` or a loop (`do`, `for` or `while`).
E 71:	illegal type specified	The type you specified is not allowed in this context.
W 72:	duplicate type modifier	Type qualifiers may not be repeated in a specifier list or qualifier list.
E 73:	object cannot be bound to multiple memories	Use only one memory attribute per object.
E 74:	declaration contains more than one class specifier	A declaration may contain at most one storage class specifier.
E 75:	'continue' outside a loop	`continue` may only appear in a loop body (`do`, `for` or `while`).
E 76:	duplicate macro parameter "*name*"	The given identifier was used more than one in the format1 parameter list of a macro definition.
E 77:	parameter list should be empty	An identifier list, not part of a function definition, must be empty.
E 78:	'void' should be the only parameter	Within a function prototype of a function that does not except any arguments, void may be the only parameter.
E 79:	constant expression expected	A constant expression may not contain a comma. Also, the bit field width, an expression that defines an `enum`, array-bound constants and `switch case` expressions must all be integral constant expressions.
E 80:	'#' operator shall be followed by macro parameter	The '#' operator must be followed by a macro argument.
E 81:	'##' operator shall not occur at beginning or end of a macro	The '##' (token concatenation) operator is used to paste together adjacent preprocessor tokens, so it cannot be used at the beginning or end of a macro body.
W 86:	escape character truncated to 8 bit value	The value of a hexadecimal escape sequence (a backslash, \, followed by a 'x' and a number) must fit in 8 bits storage.
E 87:	concatenated string too long	The resulting string was longer than the limit of 1500 characters.
W 88:	"*name*" redeclared with different linkage	The specified identifier was already declared.

Frontend

E 89:	illegal bitfield declarator	A bit field may only be declared as an integer, not as a pointer or a function for example.
E 90:	#error *message*	The *message* is the descriptive text supplied in a '**#error**' preprocessor directive.
W 91:	no prototype for function "*name*"	Each function should have a valid function prototype.
W 92:	no prototype for indirect function call	Each function should have a valid function prototype.
I 94:	hiding earlier one	Additional message which is preceded by error E 63. The second declaration will be used.
F 95:	protection error: *message*	Something went wrong with the protection key initialization.
E 96:	syntax error in #define	`#define id(` requires a right-parenthesis ')'.
E 97:	"..." incompatible with old-style prototype	If one function has a parameter type list and another function, with the same name, is an old-style declaration, the parameter list may not have ellipsis.
E 98:	function type cannot be inherited from a typedef	A `typedef` cannot be used for a function definition.
F 99:	conditional directives nested too deep	'**#if**', '**#ifdef**' or '**#ifndef**' directives may not be nested deeper than 50 levels.
E 100:	case or default label not inside switch	The `case:` or `default:` label may only appear inside a `switch`.
E 101:	vacuous declaration	Something is missing in the declaration.
E 102:	duplicate case or default label	Switch case values must be distinct after evaluation and there may be at most one `default:` label inside a `switch`.
E 103:	may not subtract pointer from scalar	The only operands allowed on subtraction of pointers is pointer - pointer, or pointer - scalar.
E 104:	left operand of operator has not struct/union type	The first operand of a '**.**' or '**->**' must have a `struct` or `union` type.
E 105:	zero or negative array size - ignored	Array bound constants must be greater than zero.
E 106:	different constructors	Compatible function types with parameter type lists must agree in number of parameters and in use of ellipsis. Also, the corresponding parameters must have compatible types.
E 107:	different array sizes	Corresponding array parameters of compatible function types must have the same size.
E 108:	different types	Corresponding parameters must have compatible types and the type of each prototype parameter must be compatible with the widened definition parameter.

Error/Warning Messages

Frontend

E 109:	floating point constant out of valid range	A floating point constant must have a value that fits in the type to which it was assigned.
E 110:	function cannot return arrays or functions	A function may not have a return type that is of type array or function. A pointer to a function is allowed.
I 111:	parameter list does not match earlier prototype	Check the parameter list or adjust the prototype. The number and type of parameters must match.
E 112:	parameter declaration must include identifier	If the declarator is a prototype, the declaration of each parameter must include an identifier. Also, an identifier declared as a `typedef` name cannot be a parameter name.
E 114:	incomplete struct/union type	The `struct` or `union` type must be known before you can use it.
E 115:	label "*name*" undefined	A `goto` statement was found, but the specified label did not exist in the same function or module.
W 116:	label "*name*" not referenced	The given label was defined but never referenced. The reference of the label must be within the same function or module.
E 117:	"*name*" undefined	The specified identifier was not defined. A variable's type must be specified in a declaration before it can be used.
W 118:	constant expression out of valid range	A constant expression used in a case label may not be too large. Also when converting a floating point value to an integer, the floating point constant may not be too large.
E 119:	cannot take 'sizeof' bitfield or void type	The size of a bit field or `void` type is not known. So, the size of it cannot be taken.
E 120:	cannot take 'sizeof' function	The size of a function is not known. So, the size of it cannot be taken.
E 121:	not a function declarator	This is not a valid function.
E 122:	unnamed formal parameter	The parameter must have a valid name.
W 123:	function should return something	A return in a non-`void` function must have an expression.
E 124:	array cannot hold functions	An array of functions is not allowed.
E 125:	function cannot return anything	A `return` with an expression may not appear in a `void` function.
W 126:	missing return (function "*name*")	A non-`void` function with a non-empty function body must have a `return` statement.
E 129:	cannot initialize "*name*"	Declarators in the declarator list may not contain initializations. Also, an `extern` declaration may have no initializer.
W 130:	operands of operator are pointers to different types	Pointer operands of an operator or assignment ('**=**'), must have the same type.

Frontend

E 131:	bad operand type(s) of *operator*	The operator needs an operand of another type.
W 132:	value of variable "*name*" is undefined	This warning occurs if a variable is used before it is defined.
E 133:	illegal struct/union member type	A function cannot be a member of a `struct` or `union`. Also, bit fields may only have type `int` or `unsigned`.
E 134:	bitfield size out of range - set to 1	The bit field width may not be greater than the number of bits in the type and may not be negative.
W 135:	statement not reached	The specified statement will never be executed.
E 138:	illegal function call	You cannot perform a function call on an object that is not a function.
E 139:	*operator* cannot have aggregate type	The type name in a (cast) must be a scalar (not a `struct`, `union` or a pointer) and also the operand of a (cast) must be a scalar.
E 140:	*type* cannot be applied to a register/bit/bitfield object or builtin/inline function	For example, the '**&**' operator (address) cannot be used on registers and bit fields.
E 141:	*operator* requires modifiable lvalue	The operand of the '**++**', or '**--**' operator and the left operand of an assignment or compound assignment (**lvalue**) must be modifiable.
E 143:	too many initializers	There may be no more initializers than there are objects.
W 144:	enumerator "*name*" value out of range	An `enum` constant exceeded the limit for an `int`.
E 145:	requires enclosing curly braces	A complex initializer needs enclosing curly braces.
E 146:	argument #*number*: memory spaces do not match	With prototypes, the memory spaces of arguments must match.
W 147:	argument #*number*: different levels of indirection	With prototypes, the types of arguments must be assignment compatible.
W 148:	argument #*number*: struct/union type does not match	With prototypes, both the prototyped function argument and the actual argument was a `struct` or `union`, but they have different tags. The tag types should match.
E 149:	object "*name*" has zero size	A struct or union may not have a member with an incomplete type.
W 150:	argument #*number*: pointers to different types	With prototypes, the pointer types of arguments must be compatible.
W 151:	ignoring memory specifier	Memory specifiers for a struct, union or enum are ignored.
E 152:	operands of *operator* are not pointing to the same memory space	Be sure the operands point to the same memory space.

Error/Warning Messages

Frontend

E 153:	'sizeof' zero sized object	An implicit or explicit `sizeof` operation references an object with an unknown size.
E 154:	argument #*number*: struct/union mismatch	With prototypes, only one of the prototyped function argument or the actual argument was a `struct` or `union`. The types should match.
E 155:	casting lvalue '*type*' to '*type*' is not allowed	The operand of the '**++**', or '**--**' operator or the left operand of an assignment or compound assignment (**lvalue**) may not be cast to another type.
E 157:	"*name*" is not a formal parameter	If a declarator has an identifier list, only its identifiers may appear in the declarator list.
E 158:	right side of *operator* is not a member of the designated struct/union	The second operand of '**.**' or '**->**' must be a member of the designated `struct` or `union`.
E 160:	pointer mismatch at *operator*	Both operands of *operator* must be a valid pointer.
E 161:	aggregates around *operator* do not match	The contents of the structs, unions or arrays on both sides of the *operator* must be the same.
E 162:	*operator* requires an lvalue or function designator	The '**&**' (address) operator requires an **lvalue** or function designator.
W 163:	operands of *operator* have different level of indirection	The types of pointers or addresses of the operator must be assignment compatible.
E 164:	operands of *operator* may not have type 'pointer to void'	The operands of *operator* may not have operand (`void *`).
W 165:	operands of *operator* are incompatible: pointer vs. pointer to array	The types of pointers or addresses of the operator must be assignment compatible. A pointer cannot be assigned to a pointer to array.
E 166:	*operator* cannot make something out of nothing	Casting type `void` to something else is not allowed.
E 170:	recursive expansion of inline function "*name*"	An _inline function may not be recursive.
E 171:	too much tail-recursion in inline function "*name*"	If the function level is greater than or equal to 40 this error is given.
W 172:	adjacent strings have different types	When concatenating two strings, they must have the same type.
E 173:	'void' function argument	A function may not have an argument with type `void`.
E 174:	not an address constant	A constant address was expected. Unlike a static variable, an automatic variable does not have a fixed memory location and therefore, the address of an automatic is not a constant.
E 175:	not an arithmetic constant	In a constant expression no assignment operators, no '**++**' operator, no '**--**' operator and no functions are allowed.

Frontend

E 176:	address of automatic is not a constant	Unlike a static variable, an automatic variable does not have a fixed memory location and therefore, the address of an automatic is not a constant.
W 177:	static variable "*name*" not used	A static variable is declared which is never used.
W 178:	static function "name" not used	A static function is declared which is never called.
E 179:	inline function "*name*" is not defined	Possibly only the prototype of the inline function was present, but the actual inline function was not.
E 180:	illegal target memory (*memory*) for pointer	The pointer may not point to *memory*.
W 182:	argument #*number*: different types	With prototypes, the types of arguments must be compatible.
I 185:	(prototype synthesized at line *number* in "*name*")	This is an informational message containing the source file position where an old-style prototype was synthesized.
E 186:	array of type bit is not allowed	An array cannot contain bit type variables.
E 187:	illegal structure definition	A structure can only be defined (initialized) if its members are known.
E 188:	structure containing bit-type fields is forced into bitaddressable area	This error occurs when you use a bitaddressable storage type for a structure containing bit-type members.
E 189:	pointer is forced to bitaddressable, pointer to bitaddressable is illegal	A pointer to bitaddressable memory is not allowed.
W 190:	"long float" changed to "float"	In ANSI C floating point constants are treated having type `double`, unless the constant has the suffix '**f**'.
E 191:	recursive struct/union definition	A `struct` or `union` cannot contain itself.
E 192:	missing filename after -f option	The **-f** option requires a filename argument.
E 194:	cannot initialize typedef	You cannot assign a value to a `typedef` variable.
F 199:	demonstration package limits exceeded	The demonstration package has certain limits which are not present in the full version.
W 200:	unknown pragma - ignored	The compiler ignores pragmas that are not known.
W 201:	"*name*" cannot have storage type - ignored	A `register` variable or an automatic/parameter cannot have a storage type.
E 202:	"*name*" is declared with 'void' parameter list	You cannot call a function with an argument when the function does not accept any (`void` parameter list).
E 203:	too many/few actual parameters	With prototyping, the number of arguments of a function must agree with the prototype of the function.

Error/Warning Messages

Frontend

W 204:	U suffix not allowed on floating constant - ignored	A floating point constant cannot have a '**U**' or '**u**' suffix.
W 205:	F suffix not allowed on integer constant - ignored	An integer constant cannot have a '**F**' or '**f**' suffix.
E 206:	'*name*' named bit-field cannot have 0 width	A bit field must be an integral constant expression with a value greater than zero.
E 212:	"*name*": missing static function definition	A function with a `static` prototype misses its definition.
W 303:	variable '*name*' possibly uninitialized	Possibly an initialization statement is not reached, while a function should return something.
E 327:	too many arguments to pass in registers for _asmfunc '*name*'	An `_asmfunc` function uses a fixed register-based interface between C and assembly, but the number of arguments that can be passed is limited by the number of available registers. With function *name* this limit was reached.

Backend

W 501:	function qualifier used on non-function	A function qualifier can only be used on functions.
E 502:	Intrinsic function '_int()' needs an immediate value as parameter	The argument of the `_int()` intrinsic function must be an integral constant expression rather than any type of integral expression.
E 503:	Intrinsic function '_jrsf()' needs an immediate value 0..3	The given number must be a constant value between 0 and 3.
W 508:	function qualifier duplicated	Only one function qualifier is allowed.
E 511:	interrupt function must have void result and void parameter list	A function declared with `_interrupt(`*n*`)` may not accept any arguments and may not return anything.
W 512:	'*number*' illegal interrupt number (0, or 3 to 251) - ignored	The interrupt vector number must be 0, or in the range 3 to 251. Any other number is illegal.
E 513:	calling an interrupt routine, use '_swi()'	An interrupt function cannot be called directly, you must use the intrinsic function `_swi()`.
E 514:	conflict in '_interrupt'/ '_asmfunc' attribute	The attributes of the current function qualifier declaration and the previous function qualifier declaration are not the same.
E 515:	different '_interrupt' number	The interrupt number of the current function qualifier declaration and the previous function qualifier declaration are not the same.
E 516:	'*memory_type*' is illegal memory for function	The storage type is not valid for this function.

Backend

W 517:	conversion of long address to short address	This warning is issued when pointer conversion is needed.
F 524:	illegal memory model	See the compiler usage for valid arguments of the **-M** option.
E 526:	function qualifier '_asmfunc' not allowed in function definition	`_asmfunc` is only allowed in the function prototype.
E 528:	_at() requires a numerical address	You can only use an expression that evaluates to a numerical address.
E 529:	_at() address out of range for this type of object	The absolute address is not present in the specified memory space.
E 530:	_at() only valid for global variables	Only global variables can be placed on absolute addresses.
E 531:	_at() only allowed for uninitialized variables	Absolute variables cannot be initialized.
E 532:	_at() has no effect on external declaration	When declared `extern` the variable is not allocated by the compiler.
W 533:	c88 language extension keyword used as identifier	A language extension keyword is a reserved word, and reserved words cannot be used as an identifier.
E 536:	illegal syntax used for default section name '*name*' in -R option	See the description of the **-R** option for the correct syntax.
E 537:	default section name '*name*' not allowed	See the description of the **-R** option for the correct syntax.
W 538:	default section name '*name*' already renamed to '*new_name*'	Only use one of the **-R** option or the renamesect pragma or use another name.
W 542:	optimization stack underflow, no optimization options are saved with #pragma optimize	This warning occurs if you use a `#pragma endoptimize` while there were no options saved by a previous `#pragma endoptimize`.
W 555:	current optimization level could reduce debugging comfort (-g)	You could have **HLL** debug conflicts with these optimization settings.
E 560:	Float/Double: not yet implemented	Floating point will be supported in a following version.

Library

<ctype.h>	isalnum, isalpha, isascii, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, toascii, _tolower, tolower, _toupper, toupper
<errno.h>	Error numbers No C functions.
<float.h>	Constants for floating-point operation
<limits.h>	Limits and sizes of integral types No C functions.
<locale.h>	localeconv, setlocale Delivered as skeletons.
<math.h>	acos, asin, atan, atan2, ceil, cos, cosh, exp, fabs, floor, fmod, frexp, ldexp, log, log10, modf, pow, sin, sinh, sqrt, tan, tanh
<setjmp.h>	longjmp, setjmp
<signal.h>	raise, signal Functions are delivered as skeletons.
<simio.h>	_simi, _simo
<stdarg.h>	va_arg, va_end, va_start
<stddef.h>	offsetof, definition of special types
<stdio.h>	clearerr, fclose, _fclose, feof, ferror, fflush, fgetc, fgetpos, fgets, fopen, _fopen, fprintf, fputc, fputs, fread, freopen, fscanf, fseek, fsetpos, ftell, fwrite, getc, getchar, gets, _ioread, _iowrite, _lseek, perror, printf, putc, putchar, puts, _read, remove, rename, rewind, scanf, setbuf, setvbuf, sprintf, sscanf, tmpfile, tmpnam, ungetc, vfprintf, vprintf, vsprintf, _write
<stdlib.h>	abort, abs, atexit, atof, atoi, atol, bsearch, calloc, div, exit, free, getenv, labs, ldiv, malloc, mblen, mbstowcs, mbtowc, qsort, rand, realloc, srand, strtod, strtol, strtoul, system, wcstombs, wctomb
<string.h>	memchr, memcmp, memcpy, memmove, memset, strcat, strchr, strcmp, strcol, strcpy, strcspn, strerror, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strstr, strtok, strxfrm
<time.h>	asctime, clock, ctime, difftime, gmtime, localtime, mktime, strftime, time All functions are delivered as skeletons.

Startup Command

as88 [*option*]...*source-file* [*map-file*]

Options

-C *file*	Include *file* before source
-D*macro*[=*def*]	Define preprocessor *macro*
-L[*flag...*]	Remove specified source lines from list file
-M[**s**|**c**|**d**|**l**]	Specify memory model
-V	Display version header only
-c	Switch to case insensitive mode (default case sensitive)
-e	Remove object file on assembly errors
-err	Redirect error messages to error file
-f *file*	Read options from *file*
-i[**l**|**g**]	Default label style local or global
-l	Generate listing file
-o *filename*	Specify name of output file
-t	Display section summary
-v	Verbose mode. Print the filenames and numbers of the passes while they progress
-w[*num*]	Suppress one or all warning messages

Functions

@function_name(*argument*[,*argument*]...)

Mathematical Functions

ABS	Absolute value
MAX	Maximum value
MIN	Minimum value
SGN	Return sign

String Functions

CAT	Catenate strings
LEN	Length of string
POS	Position of substring in string
SCP	Compare strings
SUB	Substring from a string

Macro Functions

ARG	Macro argument function
CNT	Macro argument count
MAC	Macro definition function
MXP	Macro expansion function

Assembler Mode Functions

AS88	Assembler executable name
DEF	Symbol definition function
LST	LIST control flag value
MODEL	Selected model of the assembler

Address Handling Functions

CADDR	Code address
COFF	Code page offset
CPAG	Code page number
DADDR	Data address
DOFF	Data page offset
DPAG	Data page number
HIGH	256 byte page number
LOW	256 byte page offset

Assembler Directives

Debugging

CALLS	Pass call information to object file. Used to build a call tree at link time for overlaying overlay sections.
SYMB	Pass symbolic debug information

Assembly Control

ALIGN	Specify alignment
COMMENT	Start comment lines. This directive is not permitted in IF/ELIF/ELSE/ENDIF constructs and MACRO/DUP definitions.
DEFINE	Define substitution string
DEFSECT	Define section name and attributes
END	End of source program
FAIL	Programmer generated error message
INCLUDE	Include secondary file
MSG	Programmer generated message
RADIX	Change input radix for constants
SECT	Activate section
UNDEF	Undefine DEFINE symbol
WARN	Programmer generated warning

Symbol Definition

EQU	Equate symbol to a value; accepts forward references
EXTERN	External symbol declaration; also permitted in module body
GLOBAL	Global symbol declaration; also permitted in module body
LOCAL	Local symbol declaration
NAME	Identify object file
SET	Set symbol to a value; accepts forward references

Data Definition/Storage Allocation

ASCII	Define ASCII string
ASCIZ	Define NULL padded ASCII string
DB	Define constant byte
DS	Define storage
DW	Define constant word

Macros and Conditional Assembly

DUP	Duplicate sequence of source lines
DUPA	Duplicate sequence with arguments
DUPC	Duplicate sequence with characters
DUPF	Duplicate sequence in loop
ENDIF	End of conditional assembly
ENDM	End of macro definition
EXITM	Exit macro
IF	Conditional assembly directive
MACRO	Macro definition
PMACRO	Purge macro definition

Error Messages

Warnings (W)

W 101:	use *option* at the start of the source; ignored	Primary options must be used at the start of the source.
W 102:	duplicate attribute "*attribute*" found	An attribute of an EXTERN directive is used twice or more. Remove one of the duplicate attributes.
W 104:	expected an attribute but got *attribute*; ignored	
W 105:	section activation expected, use *name* directive	Use the SECT directive to activate a section.
W 106:	conflicting attributes specified "*attributes*"	You used two conflicting attributes in an EXTERN statement directive.
W 107:	memory conflict on object "*name*"	A label or other object is explicit or implicit defined using incompatible memory types.
W 108:	object attributes redefinition "*attributes*"	A label or other object is explicit or implicit defined using incompatible attributes.
W 109:	label "*label*" not used	The label *label* is defined with the GLOBAL directive and neither defined nor referred, or the label is defined with the LOCAL directive and not referenced.
W 110:	extern label "*label*" defined in module, made global	The label *label* is defined with an EXTERN directive and defined as a label in the source. The label will be handled as a global label.
W 111:	unknown $LIST flag "*flag*"	You supplied an unknown *flag* to the $LIST control.
W 112:	text found after END; ignored	An END directive designates the end of the source file. All text after the END directive will be ignored.
W 113:	unknown $MODEL specifier; ignored	You supplied an unknown model.
W 114:	$MODEL may only be specified once, it remains "*model*"; ignored	You supplied more than one model.
W 115:	use ON or OFF after control name	The control you specified must have either ON or OFF after the control name.
W 116:	unknown parameter "*parameter*" for *control-name* control	See the description of the control for the allowed parameters.
W 118:	inserted "*extern name*"	The symbol *name* is used inside an expression, but not defined with an EXTERN directive.
W 119:	"*name*" section has not the MAX attribute; ignoring RESET	

Warnings (W)

W 120:	assembler debug information: cannot emit non-tiof expression for *label*	The SYMB record contains an expression with operations that are not supported by the IEEE-695 object format.
W 121:	changed alignment size to *size*	
W 123:	expression: *type-error*	The expression performs an illegal operation on an address or combines incompatible memory spaces.
W 124:	cannot purge macro during its own definition	
W 125:	"*symbol*" is not a DEFINE symbol	You tried to UNDEF a symbol that was not previously DEFINEd or was already undefined.
W 126:	redefinition of "*define-symbol*"	The symbol is already DEFINEd in the current scope. The symbol is redefined according to this DEFINE.
W 127:	redefinition of macro "*macro*"	The macro is already defined. The macro is redefined according to this macro definition.
W 128:	number of macro arguments is less than definition	You supplied less arguments to the macro than when defining it.
W 129:	number of macro arguments is greater than definition	You supplied more arguments to the macro than when defining it.
W 130:	DUPA needs at least one value argument	The DUPA directive needs at least two arguments, the dummy parameter and a value parameter.
W 131:	DUPF increment value gives empty macro	The step value supplied with the DUPF macro will skip the DUPF macro body.
W 132:	IF started in previous file "*file*", line *line*	The ENDIF or ELSE pre-processor directive matches with an IF directive in another file.
W 133:	currently no macro expansion active	The @CNT() and @ARG() functions can only be used inside a macro expansion.
W 134:	"*directive*" is not supported, skipped	The supplied directive is not supported by this assembler.
W 135:	define symbol of "*define-symbol*" is not an identifier; skipped definition	You supplied an illegal identifier with the **-D** option on the command line.
W 137:	label "*label*" defined *attribute* and *attribute*	The label is defined with an EXTERN and a GLOBAL directive.
W 138:	warning: *WARN-directive-arguments*	Output from the WARN directive.
W 139:	expression must be between *hex-value* and *hex-value*	
W 140:	expression must be between *value* and *value*	

Error Messages

Warnings (W)

W 141: *global/local* label "*name*" not defined in this module; made extern	The label is declared and used but not defined in the source file.
W 170: code address maps to zero page	The code offset you specified to the @CPAG function is in the zero page.
W 171: address offset must be between 0 and FFFF	The offset you specified in the @CADDR or @DADDR function was too large.
W 172: page number must be between 0 and FF	The page number you specified in the @CADDR or @DADDR function was too large.

Errors (E)

E 200: *message*; halting assembly	The assembler stops the further processing of your source file.
E 201: unexpected newline or line delimiter	The syntax checker found a newline or line delimiter that does not confirm to the assembler grammar.
E 202: unexpected character: '*character*'	The syntax checker found a character that does not confirm to the assembler grammar.
E 203: illegal escape character in string constant	The syntax checker found an illegal escape character in the string constant that does not confirm to the assembler grammar.
E 204: I/O error: open intermediate file failed (*file*)	The assembler opens an intermediate file to optimize the lexical scanning phase. The assembler cannot open this file.
E 205: syntax error: expected *token* at *token*	The syntax checker expected to find a token but found another token.
E 206: syntax error: *token* unexpected	The syntax checker found an unexpected token.
E 207: syntax error: missing ':'	The syntax checker found a label definition or memory space modifier but missed the appended semi-colon.
E 208: syntax error: missing ')'	The syntax checker expected to find a closing parentheses.
E 209: invalid radix value, should be 2, 8, 10 or 16	The RADIX directive accepts only 2, 8, 10 or 16.
E 210: syntax error	The syntax checker found an error.
E 211: unknown model	Substitute the correct model, one of s, c, d or l.
E 212: syntax error: expected *token*	The syntax checker expected to find a token but found nothing.
E 213: label "*label*" defined *attribute* and *attribute*	The label is defined with a LOCAL and a GLOBAL or EXTERN directive.
E 214: illegal addressing mode	The mnemonic used an illegal addressing mode.
E 215: not enough operands	The mnemonic needs more operands.
E 216: too many operands	The mnemonic needs less operands.

Errors (E)

E 217: *description*	There was an error found during assembly of the mnemonic.
E 218: unknown mnemonic: "*name*"	The assembler found an unknown mnemonic.
E 219: this is not a hardware instruction (use $OPTIMIZE OFF "H")	The assembler found a generic instruction, but the **-Oh** (hardware only) option or the $OPTIMIZE ON "H" control was specified.
E 223: unknown section "*name*"	The section name specified with a SECT directive has not (yet) been defined with a DEFSECT directive.
E 224: unknown label "*name*"	A label was used which was not defined.
E 225: invalid memory type	You supplied an invalid memory modifier.
E 226: unknown symbol attribute: *attribute*	
E 227: invalid memory attribute	The assembler found an unknown location counter or memory mapping attribute.
E 228: *attr* attribute needs a number	The attribute *attr* needs an extra parameter.
E 229: only one of the *name* attributes may be specified	
E 230: invalid section attribute: *name*	The assembler found an unknown section attribute.
E 231: absolute section, expected "AT" expression	An absolute section must be specified using an 'AT *address*' expression.
E 232: MAX/OVERLAY sections need to be named sections	Sections with the MAX or OVERLAY attribute must have a name, otherwise the locator cannot overlay the sections.
E 233: *type* section cannot have *attribute* attribute	Code sections may not have the CLEAR or OVERLAY attribute.
E 234: section attributes do not match earlier declaration	In an previous definition of the same section other attributes were used.
E 235: redefinition of section	An absolute section of the same name can only be located once.
E 236: cannot evaluate expression of *descriptor*	Some functions and directives must evaluate their arguments during assembly.
E 237: *descriptor* directive must have positive value	Some directives need to have a positive argument.
E 238: Floating point numbers not allowed with DB directive	The DB directive does not accept floating point numbers.
E 239: byte constant out of range	The DB directive stores expressions in bytes.
E 240: word constant out of range	The DW directive stores expressions in words.
E 241: Cannot emit non tiof functions, replaced with integral value '0'	Floating point expressions and some functions can not be represented in the IEEE-695 object format.
E 242: the *name* attribute must be specified	A section must have the CODE or DATA attribute.

Error Messages

Errors (E)

E 243:	use $OBJECT OFF or $OBJECT "*object-file*"	
E 244:	unknown control "*name*"	The specified control does not exist.
E 246:	ENDM within IF/ENDIF	The assembler found an ENDM directive within an IF/ENDIF pair.
E 247:	illegal condition code	The assembler encountered an illegal condition code within an instruction.
E 248:	cannot evaluate origin expression of org "*name*: *address*"	All origins of absolute sections must be evaluated before creation of the object file.
E 249:	incorrect argument types for function "*function*"	The supplied argument(s) evaluated to a different type than expected.
E 250:	tiof function not yet implemented: "*function*"	The supplied object format function is not yet implemented.
E 251:	@POS(,,*start*) start argument past end of string	The *start* argument is larger than the length of the string in the first parameter.
E 252:	second definition of label "*label*"	The label is defined twice in the same scope.
E 253:	recursive definition of symbol "*symbol*"	The evaluation of the symbol depends on its own value.
E 254:	missing closing '>' in include directive	The syntax checker missed the closing '>' bracket in the INCLUDE directive.
E 255:	could not open include file *include-file*	The assembler could not open the given include-file.
E 256:	integral divide by zero	The expression contains an divide by zero.
E 257:	unterminated string	All strings must end on the same line as they are started.
E 258:	unexpected characters after macro parameters, possible illegal white space	Spaces are not permitted between macro parameters.
E 259:	COMMENT directive not permitted within a macro definition and conditional assembly	This assembler does not permit the usage of the COMMENT directive within MACRO/DUP definitions or IF/ELSE/ENDIF constructs.
E 260:	definition of "*macro*" unterminated, missing "endm"	The macro definition is not terminated with an ENDM directive.
E 261:	macro argument name may not start with an '_'	MACRO and DUP arguments may not start with an underscore.
E 262:	cannot find "*symbol*"	Could not find a definition of the argument of a '%' or '?' operator within a macro expansion.
E 263:	cannot evaluate: "*symbol*", value is unknown at this point	The symbol used with a '%' or '?' operator within a macro expansion has not been defined.

Errors (E)

E 264:	cannot evaluate: "*symbol*", value depends on an unknown symbol	Could not evaluate the argument of a '%' or '?' operator within a macro expansion.
E 265:	cannot evaluate argument of *dup* (unknown or location dependant symbols)	The arguments of the DUP directive could not be evaluated.
E 266:	*dup* argument must be integral	The argument of the DUP directive must be integral.
E 267:	*dup* needs a parameter	Check the syntax of the DUP directive.
E 268:	ENDM without a corresponding MACRO or DUP definition	The assembler found an ENDM directive without an corresponding MACRO or DUP definition.
E 269:	ELSE without a corresponding IF	The assembler found an ELSE directive without an corresponding IF directive.
E 270:	ENDIF without a corresponding IF	The assembler found an ENDIF directive without an corresponding IF directive.
E 271:	missing corresponding ENDIF	The assembler found an IF or ELSE directive without an corresponding ENDIF directive.
E 272:	label not permitted with this directive	Some directives do not accept labels.
E 273:	wrong number of arguments for *function*	The function needs more or less arguments.
E 274:	illegal argument for *function*	An argument has the wrong type.
E 275:	expression not properly aligned	
E 276:	immediate value must be between *value* and *value*	The immediate operand of the instruction does only accept values in the given range.
E 277:	address must be between $*address* and $*address*	The address operand is not in the range mentioned.
E 278:	operand must be an address	The operand must be an address but has no address attributes.
E 279:	address must be short	
E 280:	address must be short	The operand must be an address in the short range.
E 281:	illegal option "*option*"	The assembler found an unknown or misspelled command line option.
E 282:	"Symbols:" part not found in map file "*name*"	The map file may be incomplete.
E 283:	"Sections:" part not found in map file "*name*"	The map file may be incomplete.
E 284:	module "*name*" not found in map file "*name*"	The map file may be incomplete.

Error Messages

Errors (E)

E 285: *file-kind* file will overwrite *file-kind* file	The assembler warns when one of its output files will overwrite the source file you gave on the command line or another output file.
E 286: $CASE options must be given before any symbol definition	The $CASE options may only be given before any symbol is defined.
E 287: symbolic debug error: *message*	The assembler found an error in a symbolic debug (SYMB) instruction.
E 288: error in PAGE directive: *message*	The arguments supplied to the PAGE directive do not conform to the restrictions.
E 290: fail: *message*	Output of the FAIL directive. This is an user generated error.
E 291: generated check: *message*	Integrity check for the coupling between the C compiler and assembler.
E 293: expression out of range	An instruction operand must be in a specified address range.
E 294: expression must be between *hexvalue* and *hexvalue*	
E 295: expression must be between *value* and *value*	
E 296: optimizer error: *message*	The optimizer found an error.
E 297: jump address must be a code address	Jumps and jump-subroutines must have a target address in code memory.
E 298: size depends on location, cannot evaluate	The size of some constructions (notably the align directives) depend on the memory address.

Fatal Error (F)

F 401: memory allocation error	A request for free memory is denied by the system. All memory has been used.
F 402: duplicate input filename "*file*" and "*file*"	The assembler requires one input filename on the command line.
F 403: error opening *file-kind* file: "*file-name*"	The assembler could not open the given file.
F 404: protection error: *message*	No protection key or not a IBM compatible PC.
F 405: I/O error	The assembler cannot write its output to a file.
F 406: parser stack overflow	
F 407: symbolic debug output error	The symbolic debug information is incorrectly written in the object file.
F 408: illegal operator precedence	The operator priority table is corrupt.
F 409: Assembler internal error	The assembler encountered internal inconsistencies.

Fatal Error (F)

F 410: Assembler internal error: duplicate mufom "*symbol*" during rename	The assembler renames all symbols local to a scope to unique symbols. In this case the assembler did not succeed into making an unique name.
F 411: symbolic debug error: "*message*"	An error occurred during the parsing of the SYMB directive.
F 412: macro calls nested too deep (possible endless recursive call)	There is a limit to the number of nested macro expansions. Currently this limit is set to 1000.
F 413: cannot evaluate "*function*"	A function call is encountered although it should have been processed.
F 414: cannot recover from previous errors, stopped	Due to earlier errors the assembler internal state got corrupted and stops assembling your program.
F 415: error opening temporary file	The assembler uses temporary files for the debug information and list file generation. It could not open or create one of those temporary files.
F 416: internal error in optimizer	The optimizer found a deadlock situation. Try to assemble without any optimization options. Please fill out the error report form and send it to Seiko Epson.

Startup Command

lk88 [*option*]...*file*...

Options

-C	Link case insensitive (default case sensitive)
-L *directory*	Additional search path for system libraries
-L	Skip system library search
-M	Produce a link map (`.lnl`)
-N	Turn off overlaying
-O *name*	Specify basename of the resulting map files
-V	Display version header only
-c	Produce a separate call graph file (`.cal`)
-e	Clean up if erroneous result
-err	Redirect error messages to error file (`.elk`)
-f *file*	Read command line information from *file*, '-' means `stdin`
-l *x*	Search also in system library `libx.a`
-o *filename*	Specify name of output file
-r	Suppress undefined symbol diagnostics
-u *symbol*	Enter *symbol* as undefined in the symbol table
-v or **-t**	Verbose option. Print name of each file as it is processed
-w *n*	Suppress messages above warning level *n*

Error Messages

Warnings (W)

W 100:	Cannot create map file *filename*, turned off -M option	The given file could not be created.
W 101:	Illegal filename (*filename*) detected	A filename with an illegal extension was detected.
W 102:	Incomplete type specification, type index = T*hexnumber*	An unknown type reference.
W 103:	Object name (*name*) differs from filename	Internal name of object file not the same as the filename.
W 104:	'-o *filename*' option overwrites previous '-o *filename*'	Second **-o** option encountered, previous name is lost.
W 105:	No object files found	No files where specified at the invocation.
W 106:	No search path for system libraries. Use -L or env "*variable*"	System library files (those given with the **-l** option) must have a search path, either supplied by means of the environment, or by means of the option **-L**.
W 108:	Illegal option: *option* (-H or -\? for help)	An illegal option was detected.
W 109:	Type not completely specified for symbol <*symbol*> in *file*	Not a complete type specification in either the current file or the mentioned file.
W 110:	Compatible types, different definitions for symbol <*symbol*> in *file*	Name conflict between compatible types.
W 111:	Signed/unsigned conflict for symbol <*symbol*> in *file*	Size of both types is correct, but one of the types contains an unsigned where the other uses a signed type.
W 112:	Type conflict for symbol <*symbol*> in *file*	A real type conflict.
W 113:	Table of contents of *file* out of date, not searched. (Use ar ts <*name*>)	The **ar** library has a symbol table which is not up to date.
W 114:	No table of contents in *file*, not searched. (Use ar ts <*name*>)	The **ar** library has no symbol table.
W 115:	Library *library* contains ucode which is not supported	Ucode is not supported by the linker.
W 116:	Not all modules are translated with the same threshold (-G value)	The library file has an unknown format, or is corrupted.
W 117:	No type found for <*symbol*>. No type check performed	No type has been generated for the symbol.

Error Messages

Warnings (W)

W 118:	Variable <*name*>, has incompatible external addressing modes with file <*filename*>	A variable is not yet allocated but two external references are made by non overlapping addressing modes.
W 119:	error from the Embedded Environment: *message*, switched off relaxed addressing mode check	If the embedded environment is readable for the linker, the addressing mode check is relaxed. For instance, a variable defined as data may be accessed as huge.

Errors (E)

E 200:	Illegal object, assignment of non existing var *var*	The MUFOM variable did not exist. Corrupted object file.
E 201:	Bad magic number	The magic number of a supplied library file was not ok.
E 202:	Section *name* does not have the same attributes as already linked files	Named section with different attributes encountered.
E 203:	Cannot open *filename*	A given file was not found.
E 204:	Illegal reference in address of *name*	Illegal MUFOM variable used in value expression of a variable. Corrupted object file.
E 205:	Symbol '*name*' already defined in <*name*>	A symbol was defined twice.
E 206:	Illegal object, multi assignment on *var*	The MUFOM variable was assigned more than once probably due to a previous error 'already defined', E 205.
E 207:	Object for different processor characteristics	Bits per MAU, MAU per address or endian for this object differs with the first linked object.
E 208:	Found unresolved external(s):	There were some symbols not found.
E 209:	Object format in *file* not supported	The object file has an unknown format, or is corrupted.
E 210:	Library format in *file* not supported	The library file has an unknown format, or is corrupted.
E 211:	Function <*function*> cannot be added to the already built overlay pool <*name*>	The overlay pool has already been built in a previous linker action.
E 212:	Duplicate absolute section name <*name*>	Absolute sections begin on a fixed address. They cannot be linked.
E 213:	Section <*name*> does not have the same size as the already linked one	A section with the EQUAL attribute does not have the same size as other, already linked, sections.
E 214:	Missing section address for absolute section <*name*>	Each absolute section must have a section address command in the object. Corrupted object file.

Errors (E)

E 215:	Section <*name*> has a different address from the already linked one	Two absolute sections may be linked (overlaid) on some conditions. They must have the same address.
E 216:	Variable <*name*>, name <*name*> has incompatible external addressing modes	A variable is allocated outside a referencing addressing space.
E 217:	Variable <*name*>, has incompatible external addressing modes with file <*filename*>	A variable is not yet allocated but two external references are made by non overlapping addressing modes.
E 218:	Variable <*name*>, also referenced in <*name*> has an incompatible address format	An attempt was made to link different address formats between the current file and the mentioned file.
E 219:	Not supported/illegal *feature* in object format *format*	An option/feature is not supported or illegal in given object format.
E 220:	page size (0x*hexvalue*) overflow for section <*name*> with size 0x*hexvalue*	Section is too big to fit into the page.
E 221:	*message*	Error generated by the object.
E 222:	Address of <*name*> not defined	No address was assigned to the variable. Corrupted object file.

Fatal Errors (F)

F 400:	Cannot create file *filename*	The given file could not be created.
F 401:	Illegal object: Unknown command at offset *offset*	An unknown command was detected in the object file. Corrupted object file.
F 402:	Illegal object: Corrupted hex number at offset *offset*	Wrong byte count in hex number. Corrupted object file.
F 403:	Illegal section index	A section index out of range was detected. Corrupted object file.
F 404:	Illegal object: Unknown hex value at offset *offset*	An unknown variable was detected in the object file. Corrupted object file.
F 405:	Internal error *number*	Internal fatal error.
F 406:	*message*	No key no IBM compatible PC.
F 407:	Missing section size for section <*name*>	Each section must have a section size command in the object. Corrupted object file.
F 408:	Out of memory	An attempt to allocate more memory failed.
F 409:	Illegal object, offset *offset*	Inconsistency found in the object module.

Error Messages

Fatal Errors (F)

F 410: Illegal object	Inconsistency found in the object module at unknown offset.
F 413: Only *name* object can be linked	It is not possible to link object for other processors.
F 414: Input file *file* same as output file	Input file and output file cannot be the same.
F 415: Demonstration package limits exceeded	One of the limits in this demo version was exceeded.

Verbose (V)

V 000: Abort !	The program was aborted by the user.
V 001: Extracting files	Verbose message extracting file from library.
V 002: File currently in progress:	Verbose message file currently processed.
V 003: Starting pass *number*	Verbose message, start of given pass.
V 004: Rescanning....	Verbose message rescanning library.
V 005: Removing file *file*	Verbose message cleaning up.
V 006: Object file *file* format *format*	Named object file does not have the standard tool chain object format TIOF-695.
V 007: Library *file* format *format*	Named library file does not have the standard tool chain **ar88** format.
V 008: Embedded environment *name* read, relaxed addressing mode check enabled	Embedded environment successfully read.

Startup Command

lc88 [*option*]...[*file*]...

Options

-M	Produce a locate map file (`.map`)
-S *space*	Generate specific *space*
-V	Display version header only
-d *file*	Read description file information from *file*, '-' means `stdin`
-e	Clean up if erroneous result
-err	Redirect error messages (`.elc`)
-f *file*	Read command line information from *file*, '-' means `stdin`
-f *format*	Specify output format
-o *filename*	Specify name of output file
-p	Make a proposal for a software part on `stdout`
-v	Verbose option. Print name of each file as it is processed
-w *n*	Suppress messages above warning level *n*

Error Messages

Warnings (W)

W 100:	Maximum buffer size for *name* is *size* (Adjusted)	For the given format, a maximum buffer size is defined.
W 101:	Cannot create map file *filename*, turned off -M option	The given file could not be created.
W 102:	Only one -g switch allowed, ignored -g before *name*	Only one .out file can be debugged.
W 104:	Found a negative length for section *name*, made it positive	Only stack sections can have a negative length.
W 107:	Inserted '*name*' keyword at line *line*	A missing keyword in the description file was inserted.
W 108:	Object name (*name*) differs from filename	Internal name of object file not the same as the filename.
W 110:	Redefinition of system start point	Usually only one load module will access the system table (__lc_pm).
W 111:	Two -o options, output name will be *name*	Second **-o** option, the message gives the effective name.
W 112:	Copy table not referenced, initial data is not copied	If you use a copy statement in the layout part, the initial data is located in rom.
W 113:	No .out files found to locate	No files where specified at the invocation.
W 114:	Cannot find start label *label*	No start point found.
W 116:	Redefinition of name at line *line*	Identifier was defined twice.
W 119:	File *filename* not found in the argument list	All files to be located must be given as an argument.
W 120:	unrecognized name option <*name*> at line *line* (inserted '*name*')	Wrong option assignment. Check the manual for possibilities.
W 121:	Ignored illegal sub-option '*name*' for *name*	An illegal format sub option was detected.
W 122:	Illegal option: *option* (-H or -\? for help)	An illegal option was detected.
W 123:	Inserted *character* at line *line*	The given character was missing in the description file.
W 124:	Attribute *attribute* at line *line* unknown	An unknown attribute was specified in the description file.
W 125:	Copy table not referenced, blank sections are not cleared	Sections with attribute blank are detected, but the copy table is not referenced. The locator generates info for the startup module in the copy table for clearing blank sections at startup.

Error Messages

Warnings (W)

W 127: Layout *name* not found	The used layout in the named file must be defined in the layout part.
W 130: Physical block *name* assigned for the second time to a layout	It is not possible to assign a block more than once to a layout block.
W 136: Removed character at line *line*	The character is not needed here.
W 137: Cluster *name* declared twice (layout part)	The named cluster is declared twice.
W 138: Absolute section *name* at non-existing memory address 0x*hexnumber*	Absolute section with an address outside physical memory.
W 139: *message*	Warning message from the embedded environment.
W 140: File *filename* not found as a parameter	All processes defined in the locator description file (software part) must be specified on the invocation line.
W 141: Unknown space <*name*> in -S option	An unknown space name was specified with a -S option.
W 142: No room for section *name* in read-only memory, trying writable memory ...	A section with attribute read-only could not be placed in read-only memory, the section will be placed in writable memory.

Errors (E)

E 200: Absolute address 0x*hexnumber* occupied	An absolute address was requested, but the address was already occupied by another section.
E 201: No physical memory available for section *name*	An absolute address was requested, but there is no physical memory at this address.
E 202: Section *name* with mau size *size* cannot be located in an addressing mode with mau size *size*	A bit section cannot be located in a byte oriented addressing mode.
E 203: Illegal object, assignment of non existing var *var*	The MUFOM variable did not exist.
E 204: Cannot duplicate section '*name*' due to hardware limitations	The process must be located more than once, but the section is mapped to a virtual space without memory management possibilities.
E 205: Cannot find section for *name*	Found a variable without a section, should not be possible.
E 206: Size limit for the section group containing section *name* exceeded by 0x*hexnumber* bytes	Small sections do not fit in a page any more.
E 207: Cannot open *filename*	A given file was not found.

Errors (E)

E 208: Cannot find a cluster for section *name*	No writable memory available, or unknown addressing mode.
E 210: Unrecognized keyword <*name*> at line *line*	An unknown keyword was used in the description file.
E 211: Cannot find 0x*hexnumber* bytes for section *name* (fixed mapping)	One of virtual or physical memory was occupied, or there was no physical memory at all!
E 213: The physical memory of *name* cannot be addressing in space *name*	A mapping failed. There was no virtual address space left.
E 214: Cannot map section *name*, virtual memory address occupied	An absolute mapping failed.
E 215: Available space within *name* exceeded by *number* bytes for section *name*	The available addressing space for an addressing mode has been exceeded.
E 217: No room for section *name* in cluster *name*	The size of the cluster as defined in the .dsc file is too small.
E 218: Missing *identifier* at line *line*	This identifier must be specified.
E 219: Missing ')' at line *line*	Matching bracket missing.
E 220: Symbol '*symbol*' already defined in <*name*>	A symbol was defined twice.
E 221: Illegal object, multi assignment on *var*	The MUFOM variable was assigned more than once, probably due to an error of the object producer.
E 223: No software description found	Each input file must be described in the software description in the .dsc file.
E 224: Missing <length> keyword in block '*name*' at line *line*	No length definition found in hardware description.
E 225: Missing <*keyword*> keyword in space '*name*' at line *line*	For the given mapping, the keyword must be specified.
E 227: Missing <start> keyword in block '*name*' at line *line*	No start definition found in hardware description.
E 230: Cannot locate section *name*, requested address occupied	An absolute address was requested, but the address was already occupied by another process or section.
E 232: Found file *filename* not defined in the description file	All files to be located need a definition record in the description file.
E 233: Environment variable too long in line *line*	Found environment variable in the dsc file contains too many characters.
E 235: Unknown section size for section *name*	No section size found in this .out file. In fact a corrupted .out file.

Error Messages

Errors (E)

E 236:	Unrecoverable specification at line *line*	An unrecoverable error was made in the description file.
E 238:	Found unresolved external(s):	At locate time all externals should be satisfied.
E 239:	Absolute address *addr.addr* not found	In the given space the absolute address was not found.
E 240:	Virtual memory space *name* not found	In the description files software part for the given file, a non existing memory space was mentioned.
E 241:	Object for different processor characteristics	Bits per MAU, MAU per address or endian for this object differs with the first linked object.
E 242:	*message*	Error generated by the object.
E 244:	Missing *name* part	The given part was not found in the description file, possibly due to a previous error.
E 245:	Illegal *name* value at line *line*	A non valid value was found in the description file.
E 246:	Identifier cannot be a number at line *line*	A non valid identifier was found in the description file.
E 247:	Incomplete type specification, type index = T*hexnumber*	An unknown type was referenced by the given file. Corrupted object file.
E 250:	Address conflict between block *block1* and *block2* (memory part)	Overlapping addresses in the memory part of the description file.
E 251:	Cannot find 0x*hexnumber* bytes for section *section* in block *block*	No room in the physical block in which the section must be located.
E 255:	Section '*name*' defined more than once at line *line*	Sections cannot be declared more than once in one layout/loadmod part.
E 258:	Cannot allocate reserved space for process *number*	The memory for a reserved piece of space was occupied.
E 261:	User assert: *message*	User-programmed assertion failed.
E 262:	Label '*name*' defined more than once in the software part	Labels defined in the description file must be unique.
E 264:	*message*	Error from the embedded environment.
E 265:	Unknown section address for absolute section *name*	No section address found in this .out file. In fact a corrupted .out file.
E 266:	%s %s not (yet) supported	The requested functionality is not (yet) supported in this release.

Fatal Errors (F)

F 400:	Cannot create file *filename*	The given file could not be created.
F 401:	Cannot open *filename*	A given file was not found.
F 402:	Illegal object: Unknown command at offset *offset*	An unknown command was detected in the object file. Corrupted object file.
F 403:	Illegal filename (*name*) detected	A filename with an illegal extension was detected on the command line.
F 404:	Illegal object: Corrupted hex number at offset *offset*	Wrong byte count in hex number. Corrupted object file.
F 405:	Illegal section index	A section index out of range was detected.
F 406:	Illegal object: Unknown hex value at offset *offset*	An unknown variable was detected in the object file. Corrupted object file.
F 407:	No description file found	The locator must have a description file with the description of the hardware and the software of your system.
F 408:	*message*	No protection key or not an IBM compatible PC.
F 410:	Only one description file allowed	The locator accepts only one description file.
F 411:	Out of memory	An attempt to allocate more memory failed.
F 412:	Illegal object, offset *offset*	Inconsistency found in the object module.
F 413:	Illegal object	Inconsistency found in the object module at unknown offset.
F 415:	Only *name* .out files can be located	It is not possible to locate object for other processors.
F 416:	Unrecoverable error at line *line*, *name*	An unrecoverable error was made in the description file in the given part.
F 417:	Overlaying not yet done	Overlaying is not yet done for this .out file, link it first without -r flag!
F 418:	No layout found, or layout not consistent	If there are syntax errors in the layout, it may occur that the layout is not usable for the locator.
F 419:	*message*	Fatal from the embedded environment.
F 420:	Demonstration package limits exceeded	One of the limits in this demo version was exceeded.

Error Messages

Verbose (V)

V 000:	File currently in progress:	Verbose message. On the next lines single filenames are printed as they are processed.
V 001:	Output format: *name*	Verbose message for the generated output format.
V 002:	Starting pass *number*	Verbose message, start of given pass.
V 003:	Abort !	The program was aborted by the user.
V 004:	Warning level *number*	Verbose message, report the used warning level.
V 005:	Removing file *file*	Verbose message cleaning up.
V 006:	Found file <*filename*> via path *pathname*	The description (include) file was not found in the standard directory.
V 007:	*message*	Verbose message from the embedded environment.

Keyword

address	Specify absolute memory address
amode	Specify the addressing modes
assert	Error if assertion failed
attribute	Assign attributes to clusters, sections, stack or heap
block	Define physical memory area
bus	Specify address bus
chips	Specify cpu chips
cluster	Specify the order and placement of clusters
copy	Define placement of ROM-copies of data sections
cpu	Define cpu part
dst	Destination address
fixed	Define fixed point in memory map
gap	Reserve dynamic memory gap
heap	Define heap
label	Define virtual address label
layout	Start of the layout description
length	Length of stack, heap, physical block or reserved space
load_mod	Define load module (process)
map	Map a source address on a destination address
mau	Define minimum addressable unit (in bits)
mem	Define physical start address of a chip
memory	Define memory part
regsfr	Specify register file for use by debugger
reserved	Reserve memory
section	Define how a section must be located
selection	Specify attributes for grouping sections into clusters
size	Size of address space or memory
software	Define the software part
space	Define an addressing space or specify memory blocks
src	Source address
stack	Define a stack section
start	Give an alternative start label
table	Define a table section

EPSON International Sales Operations

AMERICA

EPSON ELECTRONICS AMERICA, INC.

HEADQUARTERS
2580 Orchard Parkway
San Jose, CA 95131, U.S.A.
Phone: +1-800-228-3964 Fax: +1-408-922-0238

SALES OFFICE
Northeast
301 Edgewater Place, Suite 210
Wakefield, MA 01880, U.S.A.
Phone: +1-800-922-7667 Fax: +1-781-246-5443

EUROPE

EPSON EUROPE ELECTRONICS GmbH

HEADQUARTERS
Riesstrasse 15
80992 Munich, GERMANY
Phone: +49-89-14005-0 Fax: +49-89-14005-110

ASIA

EPSON (CHINA) CO., LTD.
23F, Beijing Silver Tower 2# North RD DongSanHuan
ChaoYang District, Beijing, CHINA
Phone: +86-10-6410-6655 Fax: +86-10-6410-7320

SHANGHAI BRANCH
7F, High-Tech Bldg., 900, Yishan Road
Shanghai 200233, CHINA
Phone: +86-21-5423-5522 Fax: +86-21-5423-5512

EPSON HONG KONG LTD.
20/F, Harbour Centre, 25 Harbour Road
Wanchai, Hong Kong
Phone: +852-2585-4600 Fax: +852-2827-4346
Telex: 65542 EPSCO HX

EPSON Electronic Technology Development (Shenzhen) LTD.
12/F, Dawning Mansion, Keji South 12th Road
Hi- Tech Park, Shenzhen
Phone: +86-755-2699-3828 Fax: +86-755-2699-3838

EPSON TAIWAN TECHNOLOGY & TRADING LTD.
14F, No. 7, Song Ren Road
Taipei 110
Phone: +886-2-8786-6688 Fax: +886-2-8786-6660

EPSON SINGAPORE PTE., LTD.
1 HarbourFront Place
#03-02 HarbourFront Tower One, Singapore 098633
Phone: +65-6586-5500 Fax: +65-6271-3182

SEIKO EPSON CORPORATION
KOREA OFFICE
50F, KLI 63 Bldg., 60 Yoido-dong
Youngdeungpo-Ku, Seoul, 150-763, KOREA
Phone: +82-2-784-6027 Fax: +82-2-767-3677

GUMI OFFICE
2F, Grand B/D, 457-4 Songjeong-dong
Gumi-City, KOREA
Phone: +82-54-454-6027 Fax: +82-54-454-6093

SEIKO EPSON CORPORATION
SEMICONDUCTOR OPERATIONS DIVISION

IC Sales Dept.
IC International Sales Group
421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN
Phone: +81-42-587-5814 Fax: +81-42-587-5117

S5U1C88000C Manual **I**
(Integrated Tool Package for S1C88 Family)
C Compiler/Assembler/Linker

SEIKO EPSON CORPORATION
SEMICONDUCTOR OPERATIONS DIVISION

■ **EPSON Electronic Devices Website**

http://www.epson.jp/device/semicon_e

Document code: 411391200
First issue October, 2001
Printed January, 2008 in Japan ⓁB