unSP Programming Tools User's Manual

V1.2 03/31/2008

3F, No.8, Dusing Rd., Hsinchu Science Park, Hsinchu City 30078, Taiwan, R.O.C.

Tel: +886-3-666-2118 Fax: +886-3-666-2117 Web: www.generalplus.com

Table of Content

Revision History

Revision	Date	By	Remark
1.2	2008/3/31	Summer Yi	Add SetLoadAdr and -dupq=0xXX
1.1	2008/2/27	Summer Yi	Add –sr, -delunusingobj and –listallsymbol options
1.0	2007/12/16	Summer Yi	Original

1 C Compiler

1.1 General Description

The C compiler for *unSP* platform complies with ANSI C. Files with extension named with ".c", and ".C" are treated as C source programs. The C source program is preprocessed and compiled to generate translated assembly program with file name extension ".asm".
There are two versions of C compiler for *unSP,* near compiler and far compiler, differentiated by the size of the pointer. The differences between these two compilers are:

Near compiler

- 16-bit pointer
- Contains gcc.exe, cc1.exe and cpp.exe
- Smaller code size
- Does not support global data in non-PAGE0

Far compiler

- 32-bit pointer
- Contains udocc.exe, gfec.exe, inline.exe, be.exe, cpp.exe.(cpp.exe same as Near compiler)
- Bigger code size
- Supports global data in non-PAGE0
- More convenient

1.2 Command Line Options

The command line options change the behavior of the compiler. The command line options for near compiler are listed as below:

Table 1-1

-S	Compile to assembly language.
-E	Run only the preprocessor on the named C programs.
-o *file*	Place output in file *"file".*
--help	Print a description of the command line options recognized by Compiler.

-ansi	Support all ANSI standard C programs. Turn off certain features of GCC that are incompatible with ANSI C. The "-ansi" option does not cause non-ANSI programs to be rejected gratuitously. For that, "-pedantic" is required in addition to "-ansi".
-pedantic	Issue all the warnings demanded by strict ANSI C.
-w	Inhibit all warning messages.
-Wall	Enable all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros.
-Werror	Make all warnings into errors.
-Q	Make the compiler print out each function name as it is compiled, and print some statistics about each pass when it finishes.
-D*macro*	Define macro *"macro"* with the string "1" as its definition.
-D*macro=defn*	Define macro *"macro"* as *"defn".* All instances of "-D" on the command line are processed before any "-U" options.
-U*macro*	Undefine macro *"macro".*
-gstabs	Produce debugging information for the IDE.
-Idir	Add the directory *"dir"* to the head of the list of directories to be searched for header files.
-O0	No optimization.
-O1	The compiler tries to reduce code size and execution time.
-O2	Optimize more than O1. Nearly all supported optimizations that do not involve a space-speed tradeoff are performed.
-O3	Optimize more than O2. This turns on all optimizations -O2 does. In addition, turn function in-lining on and other optimizations which may increase the code size.
-Os	Optimize for size. This enables all -O2 optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size
-mglobal-var-ram	Assign uninitialized global variables in .RAM section (default).
-mglobal-var-iram	Assign uninitialized global variables in .IRAM section.
-mpage0-maskrom	Does not generate function pointer data and switch jump table in .CODE section.
-mwarn-sec-var	Issue a warning when the compile meets a variable with user defined section.
-mISA=1.0	Generate codes for *unSP*-1.0
-mISA=1.1	Generate codes for *unSP*-1.1
-mISA=1.2	Generate codes for *unSP*-1.2 (default).
-mISA=2.0	Generate codes for *unSP*-2.0

Example 1:

In DOS command line, type:

```
gcc -S test.c -o test.asm
```

The test.c is a C source file. It generates an assembly file named test.asm.

Example 2:

```
gcc -S -gstabs test.c -o test.asm
```

The test.c is a C source file. It generates an assembly file named test.asm, which includes the debug information.

Example 3:

```
gcc -S –O2 test.c -o test.asm
```

The test.c is a C source file. It generates an optimized assembly file named test.asm.

Example 4:

```
gcc -S -O2 -gstabs test.c -o test.asm
```

The test.c is a C source file. It generates an optimized assembly file named test.asm, which includes the debug information.

The following table lists command line options for far compiler.

Table 1-2

-S	Compile to assembly language.
-E	Run only the preprocessor on the named C programs.
-o *file*	Place output in file *"file".*
--help	Print a description of the command line options recognized by Compiler.
-ansi	Support all ANSI standard C programs. Turn off certain features of GCC that are incompatible with ANSI C. The "-ansi" option does not cause non-ANSI programs to be rejected gratuitously. For that, "-pedantic" is required in addition to "-ansi".
-pedantic	Issue all the warnings demanded by strict ANSI C.
-w	Inhibit all warning messages.
-Wall	Enable all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros.
-Werror	Make all warnings into errors.
-D*macro*	Define macro *"macro"* with the string "1" as its definition.
-D*macro=defn*	Define macro *"macro"* as *"defn".* All instances of "-D" on the command line are processed before any "-U" options.
-U*macro*	Undefine macro *"macro".*
-gstabs	Produce debugging information for the IDE.

-I*dir*	Add the directory *"dir"* to the head of the list of directories to be searched for header files.
-O0	No optimization.
-O1	The compiler tries to reduce code size and execution time.
-O2	Optimize more than O1. Nearly all supported optimizations that do not involve a space-speed tradeoff are performed.
-O3	Optimize more than O2. This turns on all optimizations -O2 does. In addition, turn function in-lining on and other optimizations which may increase the code size.
-mglobal-var-ram	Assign uninitialized global variables in .RAM section.
-mglobal-var-iram	Assign uninitialized global variables in .IRAM section (default).
-mISA=1.0	Generate codes for *unSP*-1.0
-mISA=1.1	Generate codes for *unSP*-1.1
-mISA=1.2	Generate codes for *unSP*-1.2 (default).
-mISA=2.0	Generate codes for *unSP*-2.0

Note:

In Generalplus *unSP* installation directory, there is another far compiler named xgcc, containing xgcc.exe, cc2.exe and cpp0.exe. Command line options are same as udocc. If users want to use xgcc, please select and click IDE menu "Project|Settgings", choose "Far Pointer", then change CC from "$(APPDIR)\udocc" to "$(APPDIR)\xgcc".

1.3 Range and Bits of Basic Data Types

Table 1-3

Data Type	Near Compiler		Far Compiler	
	Range	Bits	Range	Bits
char	–32768 to 32767	16	–32768 to 32767	16
short	–32768 to 32767	16	–32768 to 32767	16
int	–32768 to 32767	16	–32768 to 32767	16
long int	–2147483648 to 2147483647	32	–2147483648 to	32
			2147483647	
long long	–2147483648 to 2147483647	32	-2^{63} to $(2^{63} - 1)$	64
unsigned char	0 to 65535	16	0 to 65535	16
unsigned short	0 to 65535	16	0 to 65535	16
unsigned int	0 to 65535	16	0 to 65535	16
unsigned long int	0 to 4294967295	32	0 to 4294967295	32

Data Type	Near Compiler		Far Compiler	
	Range	Bits	Range	Bits
unsigned long long	0 to 4294967295	32	0 to $(2^{64} – 1)$	64
float	32-bit IEEE 754 floating point format	32	32-bit IEEE 754 floating point format	32
double	32-bit IEEE 754 floating point format	32	64-bit IEEE 754 floating point format	64

1.4 Calling Convention

Function arguments are passed on the stack and pushed onto the stack from right to left.

1.4.1 Stack Layout

Stack frame of a function is shown as below:

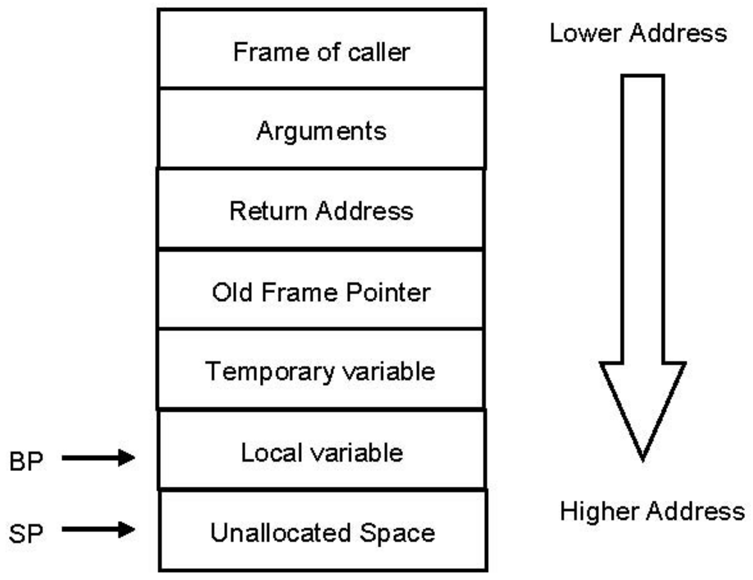

Figure 1-1

Consider the follow program:

```
 int getsum(int x, int
y) {
    int ret, t1, t2;

    t1 = x;
    t2 = y;
    ret = t1 + t2;
    return ret;
}

 int main(void)
{
    int ivalue;
    int ic = 3, id = 4;
    ivalue = getsum(ic, id);
}
```

As the program is running, the stack layout changes like this:

Step 1: Before main function calls `getsum ()`

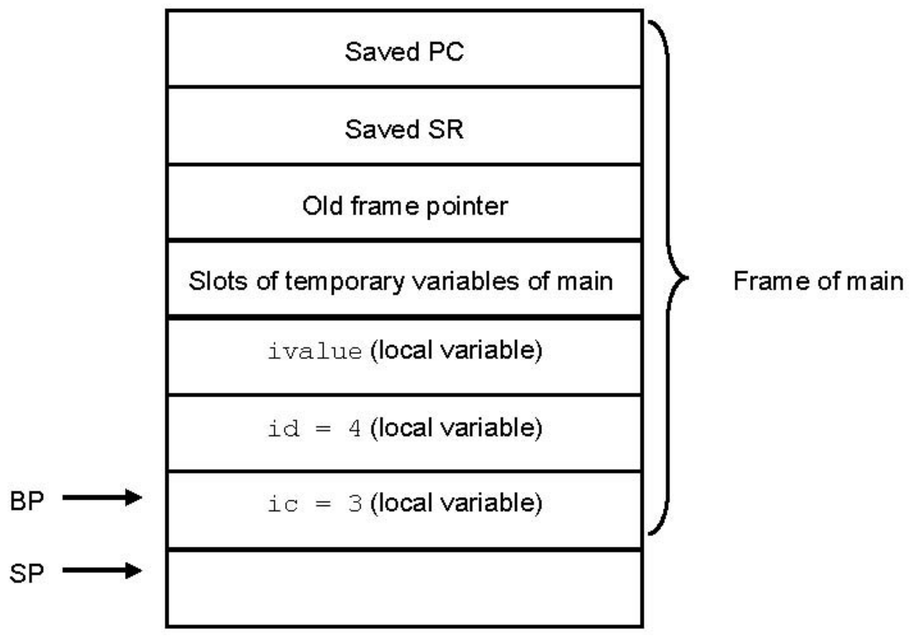

Figure 1-2

Step 2: When `getsum()` is executing

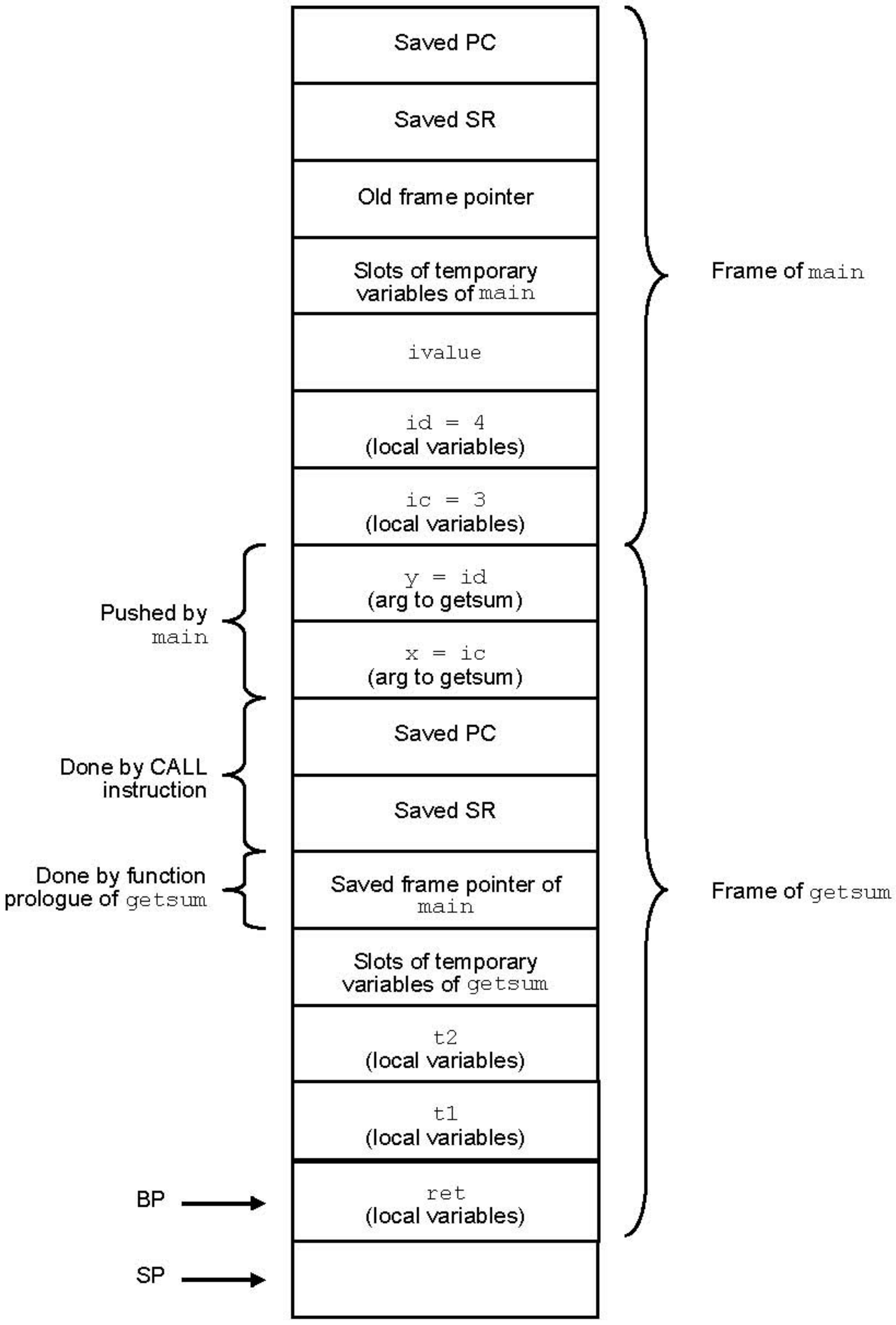

Figure 1-3

Step 3: After `getsum ()` returns

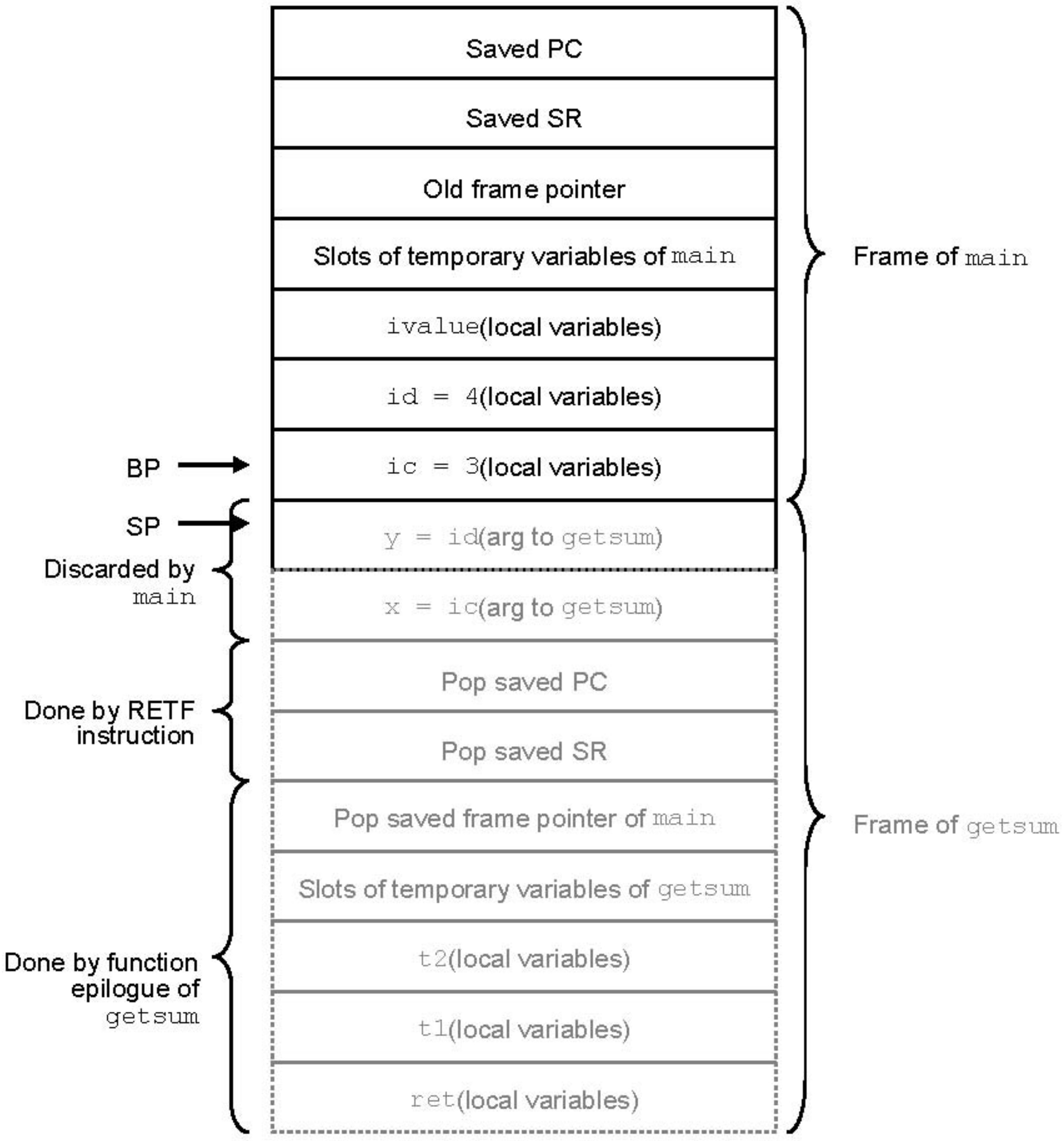

Figure 1-4

1.4.2 Convention of value passing between functions

The table below describes the principle of function arguments passing and how function value is returned:

Table 1-4

Argument Types	How arguments are passed
Standard types such as int, long and pointer	Arguments are pushed onto the stack by the caller. The right most arguments in the function argument list is pushed first.

Argument Types	How arguments are passed
Aggregate types such as struct	For near compiler: The caller duplicates a copy of the aggregate type argument and passes a pointer to this copy to the callee. For far compiler: Each element of the aggregate type argument is pushed onto the stack by the caller.
Return Types	**How function value is returned**
Standard types such as int, long and pointer	Return value is placed in the register R1 for 16-bit results, in register pair R1 and R2 with low word in R1 and high word in R2 for 32-bit results and in R1 ~ R4 for 64-bit results.
Aggregate types such as struct	The caller allocates space for the return value and passes the pointer to the allocated space as the first argument to the callee.

1.5 Special Notices for User

1.5.1 Specifying section for Variable

The keyword `__attribute__` allows you to specify special attributes of variables. Normally, the compiler places the objects it generates in specific sections. Sometimes, however, you need certain particular variables to appear in special sections, the `section` attribute specifies that a variable (or function) lives in a particular section.

Example 1:

```
int iarray1[10] __attribute__((section(".IRAM")));
```

Variable "`iarray1`" will be placed in "`. IRAM`" section by the compiler.

Example 2:

```
int iarray2[ 10] __attribute__((section (" . ISRAM" ))) = { 1, 2, 3, 4, 5} ;
```

Variable "`iarray2`" will be placed in "`.ISRAM`" section by the compiler and the initial value of `iarray2` is 1, 2, 3, 4 and 5.

1.5.2 The volatile Qualifier

Volatile objects should not participate in optimizations.

For example:

Read data from address 0x1000 until there is one bit of the data is not one.

Consider the following program fragment, which no volatile qualifier is used before PORTA:

```
#define PORTA 0x1000
int main (void)
{
   while (* (int * ) (PORTA) & 0x10) ;
}
```

Optimizer of the compiler will move the loop-invariant code outside the loop as follows, so, the read data action will only implement only once. The optimized program is similar to the following:

```
int main (void)
{
    int t1 = * (int * ) (PORTA) ;
    while (t1 & 0x10)
     /* Empty * /;
}
```

Now, Re-write the example with volatile qualifier as:

```
#define PORTA 0x1000
int main (void)
{
   while (* (volatile int * ) (PORTA) & 0x10) ;
   return 0;
}
```

Volatile qualifier would not permit compiler to move the loop-invariant code outside the loop. So, Read data action will continue implementing until there is one bit of the data is not one.

Another Example:

```
const int m = 1, n = 2;
for (i = 0; i < 100; i++)
{
    j = m * n;
}
```

Loop optimization will move "`j = m * n;` " outside the loop, while is as similar as below:

```
const int m = 1, n = 2;
j = m * n;
for (i = 0; i < 100; i++)
{
   /* Empty loop */
}
```

So, action "`j = m * n;` " will implement only once.
If you want to write some instructions to waste time definitely, you'd better use inline assembly to output "NOP" instruction as follows:

```
for (i = 0; i < 10000;
i++) asm("nop");
```

1.5.3 Function Prototype

Function declaration has two form called prototype form and traditional form.

```
// Prototype form
int f(int x, long y)
```

```
{
}
// Traditional form
int f (x, y)
    int x;
    long y;
{

}
```

The two forms of function declarations would have different effects. When calling a function, which is declared in traditional form, if the parameter is a const, it will be passed as int type to the function. When calling a function which is declared in prototype form, if the parameter is a const, it will be translated to the specified type firstly and then passed to the function. So, adopting prototype form is recommended.

Consider the following program which adopts traditional form of function declaration:

```
void set(value)
    long value;
{
    long x = value;
}

int main(void)
{
   set(0x123);
   set((long)(0x123));
}
```

Compiler will produce incorrect assembly code like this:

```
R1 = 291              // Only move 0x123 to
stack PUSH R1 to [ SP]
CALL _set             // call _set
SP = SP + 1
R1 = 291
R2 = 0
PUSH R1, R2 to [SP]   // Move 0x00000123 to stack
CALL _set             // call _set
```

When function `set` is firstly called with "`set (0x123) `", 0x123 is treated as an integer. Only 0x123 is pushed to the stack. When function `set` is secondly called with "`set ((long)(0x123) )`", 0x123 is translated to a long integer and 0x00000123 is pushed to the stack.
Now, re-write the example which adopts prototype form of function declaration:

```
void set(long value)
{
    long x = value;
}
int main(void)
{
    set(0x123);
    set((long)(0x123));
  }
```

Compiler will produce correct assembly code like this:

```
R1 = 291
R2 = 0
PUSH R1, R2 to [SP] // Move 0x00000123 to stack
CALL _set // call _set
SP = SP + 2
R1 = 291
R2 = 0
PUSH R1, R2 to [SP] // Move 0x00000123 to stack
CALL _set // call _set
```

When function `set` is called for two times, 0x123 is all treated as long integer and 0x00000123 is pushed to the stack for two times.

1.5.4 Packed String Support in Near Compiler

The smallest data type the compiler supports is 16 bits. To help reduce data size, the near compiler can compress two characters into a word by using '@' as the prefix of the string declaration. For example, the following definition

```
char a[] = "@abcdefgh";
```

will be translated by the compiler to something like this:

```
_a:
    .str '@', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'.
```

After the process of assembler and linker, char array "a" will occupy five words in memory. Assuming "a" is allocated at address 0x1000, the memory map looks like this:

```
0x1000 : 62616463666568670000
           b a d c f e h g '\0'
```

Rewrite the example without the '@' prefix as:

```
char a[] = "abcdefgh";
```

After the process of assembler and linker, string "a" will occupy nine words in memory. The memory map will look like this:

```
0x1000 : 006100620063006400650066006700680000

              a    b    c d     e f      g    h '\0'
```

1.5.5 Memory Issues of Far Compiler

The size of pointer is 32 bits, hence, 4M words memory space are accessible. Code and global variables could be beyond 64K words and the work of switching DS will be done by the compiler. Stack is still placed at the first 64K of memory. The local variable would not be beyond 64K words because they are located in the stack.

1.5.6 CPP Pre-defined Macros

Table 1-5

Near Compiler	Far Compiler
unSP	unSP
GENERALPLUS	GENERALPLUS
__POINTER_16__	__POINTER_32__
__CHAR_16__	__CHAR_16__
__INT_16__	__INT_16__
__LONG_LONG_32__	__LONG_LONG_64__
__DOUBLE_32__	__DOUBLE_64__

1.5.7 Interrupt Service Routine

In near compiler, user can define ISR(Interrupt Service Routine) in C language by using the "ISR" attribute.

For example:

```
void IRQ0(void) __attribute__ ((ISR));
void IRQ0(void)
{
  /* body * /
}
```

Note:

- Before the ISR is defined, it must be firstly declared with the "ISR" attribute.
- The function name must be one of the follows:

  IRQ0, IRQ1, IRQ2, IRQ3, IRQ4, IRQ5, IRQ6, IRQ7, RESET, BREAK

  Otherwise, linker will fail to link the ISR name.
- The compiler will make the return instruction of the ISR function as RETI, not RETF.

1.5.8 I/O Function in Simulator

There are three kinds of modes for I/O function used in simulator:

- Use printf_init() to specify I/O port and output information through hardware.

- In *unSP* IDE, in Project->Setting->Hardware->Configure page, specify I/O port and relevant output file name.
- In *unSP* IDE, in Project->Setting->Hardware->Configure page, specify I/O port and I/O window.

1.6 Inline Assembly

The format of the inline assembly instruction used in C is:

```
asm("assembly instruction template" : outputs : inputs : clobbers);
```

Note:

The clobber register only can be R1, R2, R3, R4.

For simple assembly instructions, if there is no clobber information, the inline assembly instruction can be simplified as:

```
asm("assembly instruction template" : outputs : inputs);
```

1.6.1 Assembly Instruction Template

This is the main component in the inline assembly instruction. Compiler will use it to generate assembly output at the current location. For example,

```
asm (" %0 += %1" : "+r" (foo) : "r" (bar) ) ;
```

where " `%0 += %1`" is the template and " `%0`" and " `%1`" are the operands. They will be substituted with the output and input following the first colon. The number preceded by percentage designates the n-*th* parameter after the first colon. In the following example, "%0" is the designation of `"foo",` "%1" is the designation of `"bar"` and "%2" is the designation of the value 10.

```
asm (" %0 = %1 + %2" : "=r" (foo) : "r" (bar) , "i" (10) ) ;
```

In the assembly output, the assembly instruction template will be pre-appended with a tab and ended with a new line. To write several assembly instructions in one template, you have to add line separation character ('\n') between each assembly instruction. Here is an example:

```
asm("%0 += %1\n\t%0 += %1" : "+r" (foo) : "r" (bar) ) ;
```

In this example, the '\n' makes the new line and the '\t' adds a tab in the assembly output to make it prettier.

1.6.2 Operand

The operands after the first colon represent the output operands. The operands after the second colon represent the input operands. The operands after the third colon represent the clobbered operands. Clobbered operands are operands which will have their values be destroyed, discarded and clobbered after the inline assembly instruction.

If there is more than one output operand, they should be separated by commas. The same rule applies to the input and clobbered operands.

In spite of the actual operand, an operand constraint takes place; that is, a string and a pair of parentheses should enclose the actual operand.

1.6.3 Operand Constraint

The operand constraint makes GCC recognizing which kind of value to be used in the assembly instruction template.

'r':	A value in register.
'm':	A value in memory.
'i':	An immediate value.
'p':	The name of the global variable operand.

In addition to these four constraints, there are two prefixes that should be added to output constraints.

'=':	To assign value to this operand.
'+':	The value of this operand is used during the assembly instruction template and assigned after the assembly instruction template.

1.6.4 Examples

- asm ("%0 = %1 + %2" : " = m" (foo) : "r" (bar), "i" (10));

foo and bar are local variables. GCC will assign the bar value to a register (In this example, register R1). Using the memory where foo is ([BP]) to generate the following code.

```
// GCC inline ASM
start [ BP] = R1 + 10
// GCC inline ASM
end
```

Note that the generated code cannot be assembled. The correct inline assembly instruction should be:

```
asm ("%0 = %1 + %2" : " = r" (foo) : "r" (bar), "i" (10));
And the generated code is:
// GCC inline ASM start
R1 = R4 + 10
```

```
// GCC inline ASM end
 int a;
 int b;
 #define SEG(A,B) asm("%0 = seg %1" : " = r" (A) : "p" (&B));
 int main(void)
 {
         int foo;
         int bar;
         SEG(foo, a);
         SEG(bar, b);
         return foo;
 }
```

The SEG macro is used to get the segment address of global variables.

- asm ("%0 + = %1" : "+r" (foo) : "r" (bar));

In this example, the foo value is used before assigned. The constraint should be " + r", not " = r". Or GCC will make an incorrect decision during data flow analysis.

2 Assember

2.1 Run Assembler

To run the assembler, type: xasm16

User should see the prompt as follows after running xasm16:

Generalplus unSP Assembler - Ver. 1.5.0 (Build:2)

Usage:

xasm16 [-d] [-tn] [-lFileName] [-oFileName] [-iPathName] [-eMaxErrNum] [-sr] FileName

-d	Generate source level debug information.
-t1	Specify unSP Instruction Set Architecture 1.0 (default).
-t2	Specify unSP Instruction Set Architecture 1.1.
-t3	Specify unSP Instruction Set Architecture 1.2.
-t4	Specify unSP Instruction Set Architecture 2.0.
-t5	Specify unSP Instruction Set Architecture 1.3.
-e MaxErrNum	Set the maximum number of error, default 20.
-i PathName	Set an included path of PathName.
-l FileName	Generate listing in the file of FileName or in default file.
-o FileName	Generate obj in the file of FileName.
-sym	Export symbols that are in library to symbol file.
-pc = x	specify x as the leading character of packed string(default x=@) x=NULL denote disabling packed string.
-be	Use big-endian in packed string(default -le).
-le	Use little-endian in packed string.
-sr	mask the warning about SR Regester
-dupq=0xXX	specify XX as the value of ? in dup(?)(default y=0xFF). if XX is greater than 0xFF, it will report warning and it will truncate to 0x00~0xFF automatically.

Table 2-1

Item	Description
-d	The assembler generates debug information for the assembly files. Generally, it is used only for assembly files that are written in assembly directly by users.

-e MaxErrNum	Set the maximum number of error, default 20.
-i PathName	Set an included path of PathName.
-l FileName	With this option, it generates listing in the file of FileName or in default file.
-o FileName	Generate obj in the file of FileName. The default extension is **.obj.**
FileName	Set the assembly filename. The default extension is **.asm.**
-sr	mask the warning about SR Regester
-dupq=0xXX	specify XX as the value of ? in dup(?)(default y=0xFF).

Example:

In DOS command line, type:

```
xasm16 -d -l test.lst -o test.obj test.asm
```

The test.asm is an assembler file. It generates a list file with name of test.lst and an object file with name of test.obj, which includes the debug information.

2.2 Filename extension

The following table shows the default filename extensions if users omit them.

Table 2-2 Filename extensions

	Compiler
.c	The source file written in the C language and it should be inputted to the compiler.
.h	The header file; and it should be inputted to preprocessor.
	Assembler
.asm	Input to the assembler
.obj	Output from the assembler
.lst	Listing file
	Linker
.obj	Input to the linker
.lib	Library file
.tsk	Executable object code file
.s37	Motorola s37 file
	Librarian
.obj	Input file to the librarian

.lib	Output file from the librarian

2.3 Assembly Language Syntax

2.3.1 Number Base

The default number is based on **decimal** (10) in assembler. The prefix, $, designates a Hex **decimal.** The following table indicates the numerical bases available in assembler. Apply them in the suffixes of numbers.

Table 2-3 Number base

Binary	**B**
Octal	**O** or **Q**
Decimal	**D** or no base designation
Hex	**H** or **0x** as prefixes
ASCII String	Double or single quotes, i.e. **"5"** or **'5'**

2.3.2 Argument Syntax

String

A *string* argument for any directive must be enclosed in double apostrophes (") unless the directive syntax description specifies otherwise.

Value

A numeric *value* directive argument is treated as if it is in the current overall number base (the default is 10). If you prefer a number to be a different base, you must use a base suffix/prefix.

2.3.3 Operators

The valid calculation and comparison operators and their priorities are listed in ***Appendix B.***

2.3.4 Program Comments

A comment line must start with a double-slash (//) or use the COMMENT directive. A comment can follow an instruction on the same line, but it must start with the double-slash.

2.3.5 Labels

All label names are case-sensitive. The length of a non-local label can have any numbers of characters and numbers, but only 32 are significant. A label can start in any column and its name must end with a colon. All label names must start with an alphabetic character. Non-alphanumeric character cannot be used for labels except the underscore, _.

2.3.6 Local Labels

Local labels are used like non-local labels, but the definition of a local label is valid only within its own "local area", one bounded by labels which keep their definition throughout the entire program. Because of this "local area only" referencing, you can reuse their names when a program passes from one local area to the next. In the examples, LABEL1, LABEL2, and LABEL3 are global.

```
LABEL1:              or          LABEL1:
?a: NOP;                   a?:         NOP;
?b: JMP ?a;                b?:         JMP a?;
    JMP ?b;                            JMP b?;

LABEL2:              or          LABEL2:
?a: NOP;                   a?:         NOP;
?b: JMP ?a;                b?:         JMP a?;
    JMP ?b;                            JMP b?;

LABEL3:              or          LABEL3:
?a: NOP;                   a?:         NOP;
?b: JMP ?a;                b?:         JMP a?;
      JMP ?b;                          JMP b?;
```

- Each local label has a different definition when referenced in a different local area.
- ?a is not the same as a?.
- A local label can have up to 32 characters. Never use operators like + in a local label. It is safest to follow the non-local rules for all label names.
- The directives VAR, SECTION or ENDS will not terminate local labels.
- The assembler normally identifies a local label by a question mark (?) placed prefix or suffix.
- A local label must start with an alphabetic character or question mark (?).

2.3.7 High Word and Low Word Address

To get the high word of a 32-bit address, use **SEG.** This allows b16 through b21 to be used as code segment value. To get the low word of a 32-bit address, use **OFFSET.** This allows b0 through b15 to be used as offset value (See ***APPENDIXB -Assembly Time Operators).***

2.3.8 Byte Address (Applied for *unSP-1.3)*

To get the high word of a 32-bit byte address, use BYTE_**SEG**. This allows b16 through b22 to be used as code segment value in byte domain. To get the low word of a 32-bit byte address, use BYTE_**OFFSET.** This allows b0 through b15 to be used as offset value in byte domain. (See ***APPENDIXB -Assembly Time Operators).***

2.3.9 Upper and Lower Case

Assembler directives are not case sensitive. You can type them in lower or upper case, or in a combination of both. **All labels,** including macro name, struct name, struct variable name, section name and procedure name are case-sensitive.

2.4 Assembler Directives

Directives control the workings of the assembler, and must not be confused with the processor's assembly language instructions.

2.4.1 Syntax

In *unSP,* a directive can start in any position. A directive in any position may start with a **decimal** point to distinguish it from an instruction. A bracketed field or argument is optional. If an argument has double brackets, the argument is optional, but its syntax requires the inner set of brackets. For example, *[[count]]* is an optional argument, but when used, it must be entered as *[count].*
The directive can be divided into five types: Definition, Storage, Storage definition, and Conditional and assembler mode.

Types and purposes of assembler directives have been listed in the following table.

Table 2-4 Types and purposes of Assembler directives

Types	Usage	Examples
Definition	Used for defining following Contents: • Define the start and the end of a procedure or a macro; • Property, range and structure of the data used in procedure; • Define property of code or data • Exit from a macro	• PROC…ENDP; MACRO…ENDM • DEFINE, VAR, PUBLIC, XTERNAL, STRUCT…ENDS • CODE, DATA, IRAM, ORAM, T E X T … • MACEXIT
Storage	Store data as specified attribute.	DB, DW, DD, FLOAT, DOUBLE, END
Storage definition	Work with DW, FLOAT, DD, and DOUBLE to store a number of values.	DUP
Conditional	Assemble instruction conditionally	IF…ELSE…ENDIF; IFMA, IFDEF, IFNDEF

Types	Usage	Examples
Assembly Mode	• Include a file in the assembly code. • Create a user-defined section. • Enable a multi line comment. • Switch assembly mode	• INCLUDE • SECTION • COMMENT • EXTERNAL_ON, EXTERNAL_OFF, PUBLIC_ON, PUBLIC_OFF, LINKLIST_OFF, SYMBOLS_OFF

2.4.2 Notation

Bank	Page unit in memory
ROM	Read-only memory
RAM	Random-access memory
Label	Routine label
Value	Constant value
IEEE	A kind of standard real number expression in the form of exponent
Variable	Name of variable
Number	Number of data
ASCI I	The ASCII code of numerical value and symbol
Argument#	The parameter serial number in parameter table
Filename	The name of a file
[]	Optional item

2.4.3 Assembler Directives

■ **CODE**

Group: Definition

Function: Switch to predefined CODE section

Syntax: ***.CODE***

Note: All the following storage data and instructions will be stored in this section. The assembler should always be in relative mode when assembling executable instructions. At link time, the CODE section cannot across bank and only be assigned ROM address. All sections with same name (attribute) CODE are assigned ROM address separately (see ***APPENDIX B-Sections***).

Example:

```
.CODE
.PUBLIC _main
_main: .PROC
CALL Initialize;
CALL Work;
RETF;
.ENDP
```

- **COMMENT**

Group: Assembly Mode

Function: Enable a multi line comment

Syntax: ***.COMMENT terminated_char***

message terminated_char

Note: User can write blocks of comments without starting with a double–slash (//) every line. Assembler will treat everything between the chars as a comment block.

Example:

```
.COMMENT @
This is a demo for
Block comment @
```

- **CTOR**

Group: Definition

Function: Switch to predefined CTOR section

Syntax: ***.CTOR***

Note: This section simulates constructor function in C++ language. All storage data and instructions will be stored in this section. The assembler should always be in relative mode when assembling executable instructions. At link time, the CTOR section cannot cross bank and only be assigned ROM address in BANK 0. All sections with same name (attribute) of CTOR are assigned ROM address separately (See ***APPENDIX B-Sections***).

Linker uses two build-in symbols _ctor_start and _ctor_end to denote the begin address and the end address of CTOR section.

Example:

```
.CTOR
InitValue: .PROC
r1 = [0x7000];
retf;
.ENDP
```

- **DATA**

Group: Definition

Function: Switch to predefined DATA section

Syntax: ***.DATA***

Note: All the following storage data will be stored in this section. At link time, the DATA section can across bank and only is assigned ROM address. All

Example: sections with same name (attribute) DATA are assigned ROM address separately (see ***APPENDIX B-Sections***).

```
.DATA
tone_table: .DW 5,8,6,9,3,8,0;
```

- **DB**

Group: Storage

Function: Store a value in high or low byte in a 16-bit location

Syntax: ***[label:] .DB [value][,value][,...]***

Note: This directive is applied to unSP-1.3 only.

It stores $\varpi\alpha\lambda\upsilon\varepsilon\sigma$ in consecutive byte memory locations. The first value stores in the low byte of the first word and the second value stores I the high byte of the first word and so on. If odd number of values are specified, the high byte of the final word is set to 0x00. Each block defines byte data is word alignment to next label. A comma separates multiple *values*, which may be any mix of operand types. Bracket ASCII character strings with apostrophes (use two apostrophes for an embedded apostrophe).

Example:

```
Label1: .DB 0x12, 0x34, 0x56, 0x78 // Stored as 0x3412, 0x7856
Label2: .DB 0x12, 0x34, 0x56 // Stored as 0x3412, 0x0056
Label3: .DB 0x78 // Stored 0x0078
```

- **DD**

Group: Storage

Function: Store a value in a 32-bit location

Syntax: ***[label:] .DD [value][,value][,...]***

Note: Store *values* in consecutive 32-bit locations. Separate multiple *values* by commas. V*alues* may be any numeric base, but are stored as hex.

Example:

```
Label1: .DD 0x12345678 // Stored as 0x5678, 0x1234
Label2: .DD ‘PA’ // Stored 0x0050,0x0000
// 0x0041,0x0000
```

- **DOUBLE**

Group: Storage

Function: Express a value as a double

Syntax: ***label:.DOUBLE value[,value][,...]***

Note: Convert *value(s)* to double-precision floating-point number which is expressed in IEEE format. A comma separates multiple *values*. Value must be defined by floating point.

Example:

```
label1:.DOUBLE 178.125 // Stored as 0000H,0000H,4400H,4066H
label2:.DOUBLE 100.0,-178.125 // Stored as
0000H,0000H,0000H,4059H
// Stored as 0000H,0000H,4400H,C066H
```

■ DEFINE

Group: Definition

Function: Assign a variable to a value

Syntax: ***.DEFINE variable_name[value][,…]***

Note: Assign *variable* to *value*. The *value* may be another symbol or an expression. Do not forward-reference the value, as this will produce the error, "illegal forward reference".

Example:

```
.DEFINE BODY 1
.DEFINE IO_PORT 0x7016
.IFDEF BODY
R1 = 0xFFFF;
[IO_PORT] = R1;
.ENDIF
```

■ DUP

Group: Storage definition

Function: Work with DW, FLOAT, DD, and DOUBLE to store a value.

Syntax1: ***[label:] .DW number DUP(value)***

Notes 1: Reserved number of 16-bit words and stores the value in each.

Syntax2 : ***[label:] .FLOAT number DUP(value)***

Notes 2: Reserved number of 32-bit float and stores the value in each.

Syntax3: ***[label:] .DD number DUP(value)***

Notes 3: Reserved number of 32-bit long words and stores the value in each.

Syntax4 : ***[label:] .DOUBLE number DUP(value)***

Notes 4: Reserved number of 64-bit float and stores the value in each.

Example:

```
.IRAM
label1: .DW 20 DUP(0) // Reserves 20 zeroed words
label2:. DW 20 DUP(0FFh) // Reserves 20 words,
                       // storing 0FFh in each
label3: .DW 11Q DUP(20) // Reserves 9 words,
                       // storing 20 in each
label4:.DW 11 DUP(20h) // Reserves 11 words,
                       // storing 20h in each
label5:.FLOAT 20 DUP(10.982) // Reserves 20 float and stores
                           // 10.982 in each
lable6:.DOUBLE 5 DUP(5223.29) // Reserves 5 double and stores
                             // 5223.29 in each
; The .FLOAT/.DOUBLE value is expressed in IEEE format
label7:.DD 20 DUP(0) // Reserves 20 zeroed long words
label8:.DD 20 DUP(12345678H) // Reserves 20 long words and
                           // stores 12345678H in each
label9:.DW 5 DUP(?)   // Reserves 5 words
```

■ DW

Group: Storage

Function: Store a value in a 16-bit location

Syntax: ***[label:] .DW [value][,value][,...]***

Note: It stores *values* in consecutive memory locations. A comma separates multiple *values*, which may be any mix of operand types. Bracket ASCII character strings with apostrophes (use two apostrophes for an embedded apostrophe).

Example:

```
label: .DW ' Hello', 0DH;
// Stores the ASCII equivalent of the string
//"Hello" in consecutive word addresses. With a carriage return
at the
// end spaces before operands are ignored but the comma is
required.
.DW 0xFE72,0x32A6,3417H
.DW ? // Reserves one word with 0xFFFF
```

■ DTOR

Group: Definition

Function: Switch to predefined TTOR section

Syntax: ***.DTOR***

Note: This section simulates destructor function in C++ language. All storage data and instructions will be stored in this section. The assembler should always be in relative mode when assembling executable instructions. At link time, the DTOR section cannot cross bank and only be assigned ROM address in BANK 0. All sections with same name (attribute) of DTOR are assigned ROM address separately (See **APPENDIX B-Sections**).

Linker uses two build-in symbols _dtor_start and _dtor_end to denote the begin address and the end address of DTOR section.

Example:

```
.DTOR
finitValue: .PROC
r1 = [0x7000];
retf;
.ENDP
```

- **ELSE**

Group: Conditional

Function: Assemble if previous condition is false

Syntax: ***.ELSE***

Note: Defines the next statement to be assembled if a condition result is false.

Example:

```
.IF (Cond1)
[0x7016] = R1;
.ELSE
[0x7016] = R2;
.ENDIF
```

- **END**

Group: Storage

Function: Define the end of a program

Syntax: ***.END***

Note: Defines the end of a program or an included file.

Example:

```
.END
```

- **ENDIF**

Group: Conditional

Function: Define the end of a conditional block

Syntax: ***.ENDIF***

Note: Terminates a conditional block. Unmatched IF–ENDIF pairs will generate an error message.

Example:

```
.IF (Const1)
R1 = Const1;
.ENDIF; // Other code or data to assemble
```

- **ENDM**

Group: Definition

Function: Define the end of a macro

Syntax: ***.ENDM***

Note: Terminates a macro definition (See ***APPENDIXB-Macros***).

Example:
```
test1:      .MACRO  arg
.DW     arg;
.ENDM
```

- **ENDP**

Group: Definition

Function: Define the end of a procedure

Syntax: .ENDP

Note: Terminates a procedure whose definition starts with the PROC directive

Example:
```
test1:    .PROC
Push bp to [sp];
R2 = [0x7000];
R2 = R2 AND  0x8;
R1 = R2;
Pop bp from [sp];
Retf;
.ENDP
```

- **ENDS**

Group: Definition

Function: Define the end of a struct

Syntax: *.ENDS*

Note: Ends a struct definition (See ***APPENDIX E-Struct***)

Example:
```
Body1:  .STRUCT
  member:  .DW   10
  name:    .DW    'Drive1'
  value:    .DD     0ffcH
   .ENDS
```

- **EQU**

Group: Definition

Function: Equate a label to a value

Syntax: label: .EQU   value

Notes Equates *label* to *value*. The *value* may be another symbol or an expression. Do not forward-reference the value that will produce the error "illegal forward reference". All label's information defined by EQU will output to linker symbol file.

Example:
```
label:  .EQU  10;
```

- **EXTERNAL**

Group: Definition

Function: Declare a label that has been defined in other files

Syntax: .EXTERNAL label[,label][,...]

Note: States that each *label* is defined in another file. A comma separates multiple labels. The assembler does not support any math or logic operation involving two or more externals.

Example:
```
.EXTERNAL    num_var2, num_var3
Output:       .PROC
          R1 = num_var2;
          [0x7016] = R1;
          R1 = num_var3;
          [0x7016] = R1;
          RETF;
.ENDP
```

- **EXTERNAL_OFF**

Group: Assembly Mode

Function: Exit from EXTERNAL_ON Mode.

Syntax: .EXTERNAL_OFF

Example:
```
.EXTERNAL_ON      // Enter EXTERNAL_ON Mode
.CODE
sub1:   .PROC
CALL Encode     // Llabel Encode will become a external label
// without declaration
RETF;
.ENDP
.EXTERNAL_OFF        // Exit from EXTERNAL_ON Mode
```

- **EXTERNAL_ON**

Group: Assembly Mode

Function: Enter EXTERNAL_ON Mode.

Syntax: .EXTERNAL_ON

Notes All labels defined after EXTERNAL_ON are external labels if assembler can not find its declaration in this file.

Example:
```
.EXTERNAL_ON    // Enter EXTERNAL_ON Mode
.CODE
sub1:   .PROC
CALL Encode // Label Encode will become a external label
without
// declaration
RETF;
.ENDP
```

- **FLOAT**

Group: Storage

Function: Express a value as a float

Syntax: label: .FLOAT value[,value][,...]

Notes Convert value(s) to single-precision floating-point number that is expressed in IEEE format. A comma separates multiple ϖαλυεσ. FLOAT truncates the float's fraction if it is over six bits. Value must been defined by float point.

Example:

```
label1: .FLOAT 178.125              // Stored as 43322000H
label2: .FLOAT 100.0,125.0,-178.125   // Stored as 42c80000H
                                   // 42fa0000H, c3322000H
```

■ IF

Group: Conditional

Function: Assemble if this condition is true

Syntax: IF value

Note: If ϖαλυε does not equal zero, assemble subsequent statements. The ϖαλυε can be an arithmetic expression, a symbol, or a string.

Example:

```
.DEFINE var1 0x1
.IF   var1
 .DEFINE  var2 var1 + 0x7
 [0x7010] = R2;
.ENDIF
```

■ IFDEF

Group: Conditional

Function: Assemble if this variable is already defined

Syntax: .IFDEF variable

Note: Search the symbol table. If variable is already defined, it assembles subsequent statements. If not, ignore everything until the next ELSE or ENDIF. Variable must be defined by ".DEFINE"

Example:

```
.DEFINE  name0 0x1;
.IFDEF   name0
[0x7010] = R1;
.ENDIF
```

■ IFMA

Group: Conditional

Function: Assemble if this macro argument exists

Syntax: .IFMA argument#

Note: Applied inside of a macro. It scans the macro call line for the argument that number is specified. If it finds the argument, it assembles subsequent statements. If not, it ignores everything till the next ELSE or ENDIF. If *argument#* is 0, IFMA will not find an argument, but if the call line contains none, it will assemble subsequent statements (see ***APPENDIXB-Macro Examples***).

Example:

```
Bsd:  .MACRO PARM1,PARM2,PARM3
 .IFMA 3
  .DW PARM1,PARM2,PARM3
 .ENDIF
 .ENDM
Bsd  17, 35 ,78     // Stored as 17, 35, 78
Bsd  45, 23      // Will not be triggered, No action !
```

- **IFNDEF**

Group: Conditional

Function: Assemble if this variable is not already defined

Syntax: .IFNDEF variable

Note: Search the symbol table. If variable is not defined, it assembles subsequent statements. Otherwise, ignore everything till the next ELSE or ENDIF.

Example:

```
.IFNDEF  SPCE
NOP;
.ELSE
[0x7010] = R1;
.ENDIF
```

- **IM**

Group: Definition

Function: Switch to predefined IM section

Syntax: ***.IM***

Note: This section is only applied to unSP –2.0 that the instruction memory and data memory are partitioned. All storage data and instructions will be stored in this section. The assembler should always be in relative mode when assembling executable instructions. At link time, the IM section cannot cross bank and only be assigned ROM address in BANK 0. All sections with same name (attribute) of IM are assigned ROM address separately (See ***APPENDIX B-Sections***).

Example:

```
.IM
SetValue: .PROC
R2 = 0x7000;
retf;
.ENDP
```

- **INCLUDE**

Group: Assembly Mode

Function: Include this file in the assembly

Syntax: .INCLUDE filename

Note: Include a file in the assembly. *Φιλεναμε* may include a pathname. User must indicate the *filename* extensions. Each included file requires a separate include directive.

Example:

```
.INCLUDE  Hareware.inc
```

■ **IRAM**

Group: Definition

Function: Switch to predefined IRAM section

Syntax: .IRAM

Note: All storage variables with initial value will be stored in this section. The assembler should always be in relative mode when assembling executable instructions. At link time, the IRAM section can cross bank by using '-iramnpage0' option and be assigned ROM and RAM address separately, all sections with same name (attribute) IRAM are emerged together and assigned ROM and RAM address totally, and IRAM can also be assigned absolute address by using ".ADDR" directive (See ***APPENDIXB-Sections***).

Example:

```
.IRAM
Storage:  .dw    0x3512,0x7E123,0xFE67,0x12BA
.dw    0x78AE,0x6756,0x1200,0x65A0
.VAR      S1 = 16, S2 = 0x7816, S3 = 1267;
```

■ **ISRAM**

Group: Definition

Function: Switch to predefined ISRAM section

Syntax: .ISRAM

Note: All storage variables with initial value will be stored in this section. The assembler should always be in relative mode when assembling executable instructions. At link time, the ISRAM section cannot cross bank and be assigned ROM and RAM address (0~63) separately. All sections with same name (attribute) ISRAM are emerged together and assigned ROM and RAM address totally (See ***APPENDIXB-Sections***).

Example:

```
.ISRAM
Storage: .dw  0x3512, 0x7E123, 0xFE67, 0x12BA
.VAR   S1 = 16, S2 = 0x7816, S3 = 1267;
```

■ **IRAM_BANK0**

Group: Definition

Function: Switch to predefined IRAM_BANK0 section

Syntax: .IRAM_BANK0

Note: All storage variables with initial value will be stored in this section. The assembler should be always in relative mode when assembling executable instructions. At link time, the IRAM_BANK0 section cannot cross bank and be assigned ROM and RAM address (0~0xFFFF) separately. All sections with the same name (attribute) IRAM_BANK0 are emerged together and assigned ROM and RAM address totally (See ***APPENDIXB-Sections***).

Example:

```
.IRAM_bank0
Storage: .dw  0x3512, 0x7E13, 0xFE67, 0x12BA
```

■ **LINKLIST_OFF**

Group: Assembly Mode

Function: Tells the linker do not to relocate the assembler listing file.

Syntax: .LINKLIST_OFF

Example:

```
.LINKLIST_OFF
.CODE
Sub1: .PROC
.ENDP
```

■ LINKONCE

Group: Definition

Function: Switch to predefined LINKONCE section

Syntax: .LINKONCE

Note: This section is used to support C++ language. LINKONCE section mainly store instructions.

At link time:

1. All sections with same name (attribute) of LINKONCE and without absolute address are assigned only one block of memory space, the size of which is the largest one among the LINKONCE sections.

2. All sections with same name (attribute) of LINKONCE and with the same absolute address are assigned only one block of memory space, the size of which is the largest one among the LINKONCE sections.

3. All sections with same name (attribute) of LINKONCE and with the different absolute addresses are assigned ROM addresses separately (See ***APPENDIX B-Sections***).

Example:

```
.LINKONCE
InitValue: .LINKONCE
r1 = [0x7000];
retf;
.ENDP
```

■ MACEXIT

Group: Definition

Function: Exit from a macro

Syntax: label: .MACEXIT

Note: Exit is immediate and unconditional. MACEXIT does not let the macro terminated. It exits from within the macro, leaving the rest unexpected, and restores all conditionals to their values before the macro was invoked (See ***APPENDIXB-Macros***).

Example:

```
reserve:  .MACRO   arg1, arg2, arg3
  .IFNDEF     PORTA
   .MACEXIT
  .ENDIF
  .DEFINE SUM  arg1 + arg2 + arg3
  .ENDM
```

■ MACRO

Group: Definition

Function: Define the start of a macro

Syntax: label: .MACRO args

Note: Begins a macro definition (See ***APPENDIXB-Macros***).

Example:

```
ADD:  .MACRO arg1,arg2
R1 = arg1 + arg2;
.ENDM
.CODE
…
ADD 4, 6
// R1 content will be 10 after this instruction be executed
```

- **NB_DATA**

Group: Definition

Function: Switch to predefined NB_DATA section

Syntax: .NB_DATA

Note: All the following storage data will be stored in this section. At link time, the NB_DATA section cannot across bank and only be assigned ROM address. All sections with same name (attribute) NB_DATA are assigned ROM address separately. (See ***APPENDIXB-Sections***)

Example:

```
.NB_DATA
_ToneTable: .DW  4, 7, 1, 0, 4, 2, 9, 2, 23, 45, 0
```

- **NB_MERGE**

Group: Definition

Function: Switch to predefined NB_MERGE section

Syntax: ***.NB_MERGE***

Note: All the following storage data will be stored in this section. At link time, the NB_MERGE section cannot across bank and only be assigned ROM address. All sections with same name (attribute) NB_MERGE are assigned ROM address together (See ***APPENDIXB-Sections***).

Example:

```
.NB_MERGE
_ToneTable:  .DW 4, 7, 1, 0, 4, 2, 9, 2, 23, 45, 0
```

- **ORAM**

Group: Definition

Function: Switch to predefined ORAM section

Syntax: ***.ORAM***

Note: All storage variables without initial value will be stored in this section. The assembler should always be in relative mode when assembling executable instructions. At link time, the ORAM section cannot cross bank and only be assigned RAM address. All sections of an object file with same name (attribute) ORAM are emerged together and assigned RAM address totally. But, sections of different object files with same name ORAM will be overlapped in one project. (See ***APPENDIXB-Sections***).

Example:

```
In Sub1.asm:
…
SpeechBuf1:   .SECTION  .ORAM
 Buf1_SA:  .DW10 DUP(?)
…
SpeechBuf2:   .SECTION .ORAM
 Buf2_SA:   .DW  64 DUP(?)
In Sub2.asm:
…
SpeechBuf1;  .SECTION .ORAM
 Input_Buf:  .DW    20 DUP(?)
In linking:
SpeechBuf1 of Sub1.asm and SpeechBuf1 of Sub2.asm will share
the same memory space (its size will be 20 words), but
SpeechBuf2 will occupy another memory space.
```

- **OSRAM**

Group: Definition

Function: Switch to predefined OSRAM section

Syntax: .OSRAM

Note: All storage variables without initial value will be stored in this section. The assembler should always be in relative mode when assembling executable instructions. At link time, the OSRAM section cannot cross bank and only be assigned RAM address (0~63). All sections of an object file with same name (attribute) OSRAM are emerged together and assigned RAM address totally. However, sections of different object files with same name OSRAM will be overlapped in one project (See ***APPENDIXB-Sections***).

Example: Reference .ORAM

- **ORAM_BANK0**

Group: Definition

Function: Switch to predefined ORAM_BANK0 section

Syntax: .ORAM_BANK0

Note: All storage variables without initial value will be stored in this section. The assembler should be always in relative mode when assembling executable instructions. At link time, the ORAM_BANK0 section cannot cross bank and only be assigned RAM address (0~0xFFFF). All sections of an object file with same name (attribute) ORAM_BANK0 are emerged together and assigned RAM address totally. However, sections of different object files with same name OSRAM will be overlapped in one project (See ***APPENDIXB-Sections***).

Example: Reference .ORAM

- **PROC**

Group: Definition

Function: Define the start of a procedure

Syntax: label: .PROC

Note: Begins a procedure definition.

Example:
```
test1:     .PROC
PUSH BP, BP TO [SP];
BP = SP + 1;
R1 = [0x7015];
POP BP, BP FROM [SP];
.RETF;
.ENDP
```

- **PUBLIC**

Group: Definition

Function: Declare a label that may be used in other files

Syntax: .PUBLIC   label[,label][,...]

Note: Define each *label* as a global label that other files can reference. A comma separates multiple *labels*. The linker will resolve all global and external references.

Example:
```
.PUBLIC   sym1      // Declares label sym1 accessible to
other files
.PUBLIC   sym1,sym2  // Multiple declarations on the same
line are legal
// when separated by a comma.  The spaces are ignored.
```

- **PUBLIC_OFF**

Group: Assembly Mode

Function: Enter PUBLIC_OFF Mode

Syntax: .PUBLIC_OFF

Note: In PUBLIC_OFF mode, a label must be declared with PUBLIC that can be referred by other files.

Example:
```
.PUBLIC_OFF
.PUBLIC sym1   // Declares label sym1 accessible to other files
.RAM
sym1: .DW  ?
```

- **PUBLIC_ON**

Group: Assembly Mode

Function: Enter PUBLIC_ON Mode

Syntax: .PUBLIC_ON

Note: All labels defined after PUBLIC_ON are global labels that other files can refer to.

Example:

```
.PUBLIC_ON
.RAM
sym1: DW ? // Declares label sym1 and sym2 accessible to
other files
sym2: .DW ?
```

- **RAM**

Group: Definition

Function: Switch to predefined RAM section

Syntax: .RAM

Note: All storage variables without initial value will be stored in this section. The assembler should always be in relative mode when assembling executable instructions. At link time, the RAM section cannot cross bank and only be assigned RAM address. All sections with same name (attribute) of RAM are assigned RAM address separately (See ***APPENDIXB-Sections***).

Example:

```
.RAM
start: .DW       ?
```

- **RAM_BANK0**

Group: Definition

Function: Switch to predefined RAM section

Syntax: .RAM_BANK0

Note: All storage variables without initial value will be stored in this section. The assembler should be always in relative mode when assembling executable instructions. At link time, the RAM_BANK0 section cannot cross bank and only be assigned RAM address (0~0xFFFF). All sections with same name (attribute) of RAM_BANK0 are assigned RAM address separately (See ***APPENDIXB-Sections***).

Example:

```
.RAM_BANK0
start: .DW       ?
```

- **SECTION**

Group: Assembly Mode

Function: Create a user-defined section

Syntax: label: .SECTION   attribute [, .ADDR = value]

Note: Create a user-defined section. The *attribute* can be one of following sections.

CODE
NB_DATA
DATA
TEXT
RAM
SRAM
RAM_BANK0
IRAM
ISRAM
IRAM_BANK0
ORAM
OSRAM
ORAM_BANK0

User can use .ADDR directive to specify the linking address for this section.

For further details of this directive and for general discussion of predefined and user-defined sections (See ***APPENDIXB-Sections***).

Example:

```
section1: .SECTION .CODE  // Define a section which name is
section1
// and it has same link attribute with predefined
// section CODE.
Uart_Set:  .PROC
R1 = 0x40;
R2 = 0x1;
[P_UART_BaudScalarHigh] = R2;
[P_UART_BaudScalarLow] = R1;
RETF;
.ENDP
section2: .SECTION .DATA , .ADDR = 0x12000
// Define a section which name is section2 and it has same
// link attribute with predefined section DATA, its linking
// address is 0x12000
Pitch_Table: .DW 5, 3, 1, 6, 43, 7, 34, 9, 2, 13, 54, 6, 6,
62, 255, 2, 67, 123
```

- **SRAM**

Group: Definition

Function: Switch to predefined RAM section

Syntax: .SRAM

Note: All storage variables without initial value will be stored in this section. The assembler should always be in relative mode when assembling executable instructions. At link time, the SRAM section cannot cross bank and only be assigned RAM address (0~63). All sections with same name (attribute) of

SRAM are assigned RAM address separately (See ***APPENDIXB-Sections***).

Example:
```
SRAM
start: .DW        ?
```

- **STABD/STABF/STABN/STABS**

Group: Definition

Function: There are four directives that begin with '.stab'. All emit information, for use by IDE source level debugger.

- **STRING**

Group: Storage

Function: Store a string

Syntax: [label:] .STR ['@'] [, 'char'] [', char'] [,...]

Note: Store a string in consecutive memory locations. Separate multiple *char* by commas. *Char* may be any characters, but are stored as ASCII characters. If '@' is the leading character of the string, the string will be packed in memory.

Example:
```
.str '@', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'
// Stored as 0x6261, 0x6463, 0x6665, 0x6867
.str 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'
// Stored as 0x0061, 0x0062, 0x0063, 0x0064,
// 0x0065, 0x0066, 0x0067, 0x0068
```

- **STRUCT**

Group: Definition

Function: Define the start of a struct

Syntax: label: .STRUCT

Note: Begins a struct definition. (See ***APPENDIX E-Struct***)

Example:
```
test:  STRUCT
```

- **SYMBOLS_OFF**

Group: Assembly Mode

Function: Tells assembler not to generate symbol information in obj file

Syntax: .SYMBOLS_OFF

Note: Store a string in consecutive memory locations. Separate multiple *char* by commas. *Char* may be any characters, but are stored as ASCII characters. If '@' is the leading character of the string, the string will be packed in memory.

Example:
```
.SYMBOLS_OFF
.RAM
.VAR  S1,S2,W1,W2    // Symbol information will not add to obj
file
```

■ TEXT

Group: Definition

Function: Switch to predefined TEXT section

Syntax: .TEXT

Note: All storage data and instructions will be stored in this section. The assembler should always be in relative mode when assembling executable instructions. At link time, the TEXT section cannot cross bank and only be assigned ROM address inBANK 0. All sections with same name (attribute) of TEXT are assigned ROM address separately (See ***APPENDIX B-Sections***).

Example:

```
TEXT
GetValue: .PROC
r1 = [0x7000];
retf;
.ENDP
```

■ VAR

Group: Definition

Function: Equate a variable to a value

Syntax: .VAR variable_name[= value][,…]

Example:

```
.IRAM
.VAR s1 = 3, s2 = 0x7, s3 = 17;
.RAM
.VAR v1, v2, v3;
```

■ VDEF

Group: Definition

Function: Equate a label to a value. Value of label could be re-defined by this command.

Syntax: label: .VDEF value

Note: You can change the assignment at any point in the program. Do not use VDEF TO redefine a label defined as a variable.

Example:

```
Port: .VDEF   0x7010
[Port] = R1;
…
Port:  VDEF    0x7013
[Port] = R2;
```

■ VBTL

Group: Definition

Function: Switch to predefined VTBL section

Syntax: .VBTL

Note: This section is used to support C++ language. VTBL section mainly is used to store data. 9

to store data. 9

At link time, VTBL section has the same linking principle as LINKONCE section. 00111001

Example: 057

```
.VTBL 39
virtual_table: .DW  5, 8, 6, 9, 3, 8, 0;
```

2.5 The unSP ISA2.0 Assembly Issue

1. Illegal assembler hand coding rule: User should not modify the value of CS

   directly when modifying the SR.

Example:

Error Coding:

```
Nop                    // CS = 0 (This instruction carries out CS value)

 r1 = 0x13             // CS = 0 (This instruction carries out CS value)

 sr = sr or r1         // CS= 0x13 (This instruction carries out CS value)
```

Correct Coding:

To change the CS value, run call function, (e.g., CALL A22, CALL MR) and goto function (e.g., GOTO A22, GOTO MR).

```
nop // CS = 0 (This instruction carries out CS value)

call lable // CS = 0x13 (This instruction carries out CS value)


lable: // CS = 0x13(CS value of the current label)
nop

nop
```

2. The "CALL A22", "GOTO A22", "GOTO MR" should not follow the instructions modify the SR value, no matter what the SR value changes.

Example:

Error Coding1:

```
r1 = 0
sr = sr or
r1 call
xxxxxx
```

Error Coding2:

```
r1 = 0
sr = sr or
r1 goto
xxxxxx
```

Error Coding3:

```
r1 = 0
sr = sr or
r1 goto MR
```

Correct Coding1:

The NOP instruction is to be put between the instruction that modifies the SR value and the CALL A22, GOTO A22, or, GOTO MR instruction.

```
r1 = 0
sr = sr or r1
nop
call xxxxxx
```

Correct Coding2:

```
r1 = 0
sr = sr or r1
nop
goto xxxxxx
```

Correct Coding3:

```
r1 = 0
sr = sr or r1
nop
goto MR
```

3 Linker

3.1 General Description

The GENERALPLUS Linker links separate assembly programs, modules, files and sections into a single integrated program. It resolves external references, relocates addresses and it modifies listings to show run-time addresses and final opcodes. The linker generates all the most common used file formats, making a separate format-conversion utility unnecessary. With most output formats, the linker runs entirely in RAM, letting you link files of any size as long as the memory is enough.

3.2 Default Filename Extensions

Table 3-1

.obj	Input To The Linker
.lib	Library File
.tsk	Pure Binary Executable Code
.s37	Motorola s37
.map	Map File
.sym	Symbol Table (all formats)
.ary	Project File (For Automatic Mode)
.lik	Linking Script File

3.3 Limitations

Table 3-2

Categories	**Console**
Input files + library modules	No limit
Section number	4096
External labels	No limit
Library files to search	No limit
Section size	Depends on type of section

3.4 Linking Address

You can specify a section's link address at link time. You can also link a section **indirectly** with different load and run-time addresses, generating ROM-able code to be moved to RAM at run time.

3.5 Address Relocation Calculations

In relocating addresses, the linker adds the offset to whatever address the assembler has generated. The assembler keeps a table of attributes associated with each symbol used in the program.

A label preceding an instruction is re-locatable.

If a label is defined with an EQU directive, its operand type determines whether it is re-locatable. If the argument contains only one re-locatable token, the label is re-locatable. If it contains none, the label is not re-locatable (It may not contain more than one. If it does, the assembler generates an error message.).

3.6 Global Symbols

To find library global, the linker searches tables in the order of:

1. Input Files
2. Library Global Names

3.7 Output Priorities

The linker has two ways to determine symbol and code file output formats. The priority is:

1. Linker Options
2. Linker Defaults (code files with s37)

3.8 Output Formats

The linker can output the Microtek or ADHighLevel symbol table and code file with s37 or tsk (pure binary executable) format. Formats are described in Linker-Symbol Table.

3.9 Operating Instruction

You can invoke the linker in Prompt, Command Line, Enhanced Data File, and Automatic mode. The load map, an alphabetic global symbol cross-reference list and all link errors can be saved on disk. The recommended operating mode is automatic mode. In this mode, you can name the object files, library files and the path to search for the object files or library files to be linked in a data file with extension of .ARY. You can also indicate the sections address in a data file with extension of .Lik. With these data file, the linker will link those sections to user specified address. Other sections will be linked automatically according to the section's default attribution. In **Enhanced Data File** mode, you may link each of your files in one of the following ways:

- **Indirect linking**

  Useful if your program resides in ROM, but the data has initialized values that will be changed as the program runs. You need a start-up routine (the start-up routine is provided by the GENERALPLUS. With name, startupD.obj/startupR.obj, you may include this file at link time) to move it from ROM to RAM. It stacks the data normally in ROM, but links it to a run-time address.

- **Offsetting sections**

  Allow you to link a section at a specific address.

  In all modes, **<Ctrl>C** terminates the linker and returns to the operating system.

3.10 Prompt mode

To run the linker, type: **xlink16**<cr>.

Input Body Name

You must input one body name.

Input Body File Name

You must input one body file name.

- *Input*

  You will see the prompt **Input Filename:**

  Type the input filename and press **<cr>** (the default extension is **.obj.).** The linker opens the file, then prompts another **Input Filename:**

  If there is no more input file, enter **<cr>.**

- *Output*

  You will see the prompt **Output Filename:**

  There is only one output file. Enter its name and press **<cr>.**

If you type **<cr>** only, the linker gives the output file the same name as the first input file (except for the start-up routine file). The extension depends on the output format you select (See ***Linker-Default Filename Extension.)***

- *Libraries* The next prompt is **Library**

  Filename:

  Enter the library filename and press **<cr>.** You can omit an extension since the linker looks automatically for files with the extension of .lib. Type only **<cr>** if no more library files.

- *Options* The linker always prompts
  for options. Options (D,C,M,X,3,
  <CR> = Default):

  Enter the options you want (see ***Linker-Prompt mode options)*** and press **<cr>.** The options let you set load map destination and code and symbol file formats (See ***Linker-Symbol Tables).***

- *Offsets*

  You will see the following prompt:

  Enter Hex ROM Offset For '(section name)' in ' (input file name)':

  Enter the offset, and press **<cr>** (See ***Linker-Offsets).*** Then prompt:

  Enter Hex RAM Offset For '(section name)':

  If you wish that the prompted section's loading address is the same as its running address, you can press **<cr>.** If the input files and library files have multiple sections, the linker will prompt for each section's offset.

3.11 Prompt Mode Options

The option field creates a load map, specifies its destination, and selects output file and symbol table formats.

The option prompt is:

Options (D, C, M, X, 3 <CR> = Default): (see following table for the prompt modes options.)

Options are not case-sensitive. The default settings are:

Load Map	**none**
Symbol & Code Files	**processor's defaults**

At any time, only **three** options can be active:

- One Load Map destination **(D).**
- One Symbol Table format **(C** or **M).**
- One Code File format **(X** or **3).**

If you enter more than one option in a category, only the last one is effective.

Table 3-3

Prompt Modes Options Table	
Option	**Description**
-D	Create a disk file containing all link errors, an alphabetical global symbol table and the load map. This file has the same name as the linker output file with the extension of **.map.**
-C	Create a High Level Symbol Table for assembly language source display.
-M	Create a Microtek Symbol Table. The file includes all symbols.
-X	Generate a pure binary Executable output file.
-3	Generate a Motorola s37 output file.
<cr>	Generate the processor's default output format.

3.12 Command Line Mode

You can invoke the linker from the command line. The command line form is as follows (bracketed arguments are not required):

Xlink16 -c file1 [-loffs] [file2 [...]] [-ofile] [-Lfile] [...] [-options] -body bodyname

- *-c*

  Precedes the first input filename. It sets the linker to command line mode.

- *file1*

  You must specify at least one input filename.

- *-l*

  (Lower case L) precedes an optional offset for each section in the input file (i.e. if there are 12 sections, there should be a **-l** entry for each section which you want to specify an offset). You can prefix an offset with the (-) operator. **–1-nnnn** is valid offsets. (See

  Linker-Offsets).

- *file2*

  Second input file. You may link as many files as you like with the same syntax for filenames and section offsets as for the first input file.

- *-o*

  Precedes an output filename. If you omit an output filename, the linker will create a default output file with the same name as the first input file. The extension depends on the output file format.

- *-L*

  (Upper case L) introduces the name of each library file to be included in the link. You do not need to specify an extension since the linker looks automatically for files with the extension **.lib.**

The final **minus sign (-)** prefixes the options list (There must be no separating spaces.). If the command line omits options, the linker will prompt for them.

3.13 Enhanced Data File Mode

Enhanced Data File mode is a powerful way to operate the linker. A set of **keywords** provides greater flexibility than the prompt modes. You can link directly or indirectly to addresses of your own chose: define an address for a section, specify options in any order and link sections in almost any order, redefine the name for a section (you cannot locate a file **after** one not yet located) in a data file with extension of .Lik. The file has strict but straightforward syntax.

3.13.1 Command Usage

xlink16 file1.lik -body name [-bfile file2] [-rfile file3] [–novec] [–noitbl] [-nobdy] [-tsklen len] [-initdata] [-iramnpage0] [-tskMaxUsed] [-listallsymbol] [-delunusingobj]

file1:	Required. You must specify .lik filename.
name:	Specify body name.
file2:	Option. Specify body filename. (Default is body.dat)
file3:	Option. Specify external symbol file for reference (Generated by Xlink16).
novec:	Option. Not generate interrupt vector table in executive file.
noitbl:	Option. Not generate initial table (for IRAM/ISRAM section) in executive file.
nobdy:	Option. Not check -body and -bfile match or not.
len:	Specify the size of .tsk file.
initdata:	Option. Specify that initial table can be placed at nonpage0. iramnpage0:

Option. Specify that IRAM can cross page and use new startup code.

TskMaxUsed: Option. The TSK file only stores the useful part. So the size of TSK file is saved as small as possible, because the spared '0' will not be appended to the TSK file.

listallsymbol: Option. Display all SYMBOL in .map file, default only display PUBLIC SYMBOL.

-delunusingobj: Option. Not link the .obj file, which is not used.

3.13.2 Keywords

Keywords are not case sensitive. They are described in the order in which they appear.

Table 3-4

Keyword	Modifiers
Version:	*[pseudo-parameter]*
[options :]	**tsk, map, adhighlevel, microtek, m37**
Obj :	*Filename[, filename...]*
Output :	*Filename*
Lib :	*Filename[, filename...]*
Define	old_section_name **in** *object (module)* **of** *library* **with** new_section_name
*Align	**in** *object (module)* **of** *library* **with** value
Locate :	**in** *object (module)* **of** *library* **after** *section* **at** *address*

Keyword	Modifiers	
	linkafter *section*	**linkat** *address*
	common	
	reference	

Note. Align command can work only for automatic mode.

In this table, and in the following syntax definitions, the colon after a keyword is for clarity only and is optional.

■ Version

This **must** be the **first** keyword in the file.

Function: Inform linker to use Enhanced Data File mode.

Syntax: version: [pseudo-parameter]

Example: version: 1.44 dated 07/03/01

Comments may precede or follow **version** on the same or on separate lines, but it must be the first **keyword** in the file. **Version** is insensitive to trailing characters on the same line. You can add a version number, date, or other information as a pseudo-parameter.

□ Options

Function: Control the creation and destination of a load map, the formats of output symbol and code files.

Syntax: ***options: option[, option ...]***

Example: options: tsk, map

A link accepts up to **three** options parameters: a load map, a symbol table format, and a code file format. Options parameters are insensitive to case and to leading, trailing and internal white space. A comma separates each option from the next. If more than one parameter is entered in a category, or if you change a parameter later in the file, the linker acts on your last choice.

The following table describes the options parameters in detail.

Table 3-5

Link Options Parameters	
Load Map and Destination	
Map	Create **.map** disk file containing link errors, section summary, and load map.
Symbol Table Formats	
Adhighlevel	High Level (for assembly language source display).
Microtek	Microtek - all symbols.
Code File Formats	
Tsk	Executable (pure binary)
m37	Motorola s37

- **Obj**

Function: Names the input file(s) to be linked. The default object extension is **.obj**, but you can specify otherwise.

Syntax: ***obj : "filename"[," filename" ...]***

Example:

```
obj : "mytest", "yourtest"
obj : "mytest.obj", "yourtest.obj"
```

- **Output**

Function: Names the output file. The default extension is the processor's default code file format extension, but you can specify otherwise.

Syntax: ***output : "filename"***

Examples:

```
output : "ourtest"
output : "ourtest.ext"
```

- **Lib**

Function: Names the library file(s). The default extension is **.lib**.

Syntax: ***lib : "libfile"[," libfile" ...]***

Examples:

```
lib : "Cmacro.lib", "Printfl.ib"
```

- **Define**

Function: Redefine a section in object (module) of library with a new section name.

Syntax: ***define: old_section_name [In "object(module)" [of "library"]] with new_section_name***

Examples:

```
define: MySec1 in "speech.obj" with SpeechBuf// Redefine
Section "MySec1" of speech.obj with new name "SpeechBuf"
define:WrkArea in "div.obj" of "math.lib" with Common // Redefine
Section "WrkArea" of div.obj with new name "Common"
```

- **Align**

Function: Align a section in object (module) of library with a numeric value.

Syntax: ***align: section [In "object(module)" [of "library"]] with value***

Examples:

```
align: section1 of "sub_proc.obj" with 4
// align section "section1" of sub_proc.obj with 4
align: section2 in "div.obj" of "math.lib" with 6
```

- **Locate**

Function: Locates a section in object (module) of library at (linkat) a numeric address or after

(linkafter) a section already located.

Syntax: ***locate : section [In "object(module)" [of "library"]] at address, after section name***

[linkat address, linkafter section name]

[common, reference]

Examples:

```
locate : section1 in "sub1.obj" at 1234h
locate : section2 in "sub2.obj" after CODE // section2 will
be
linking after CODE
locate : section3 in "test1.obj" at 8000
locate : section4 in "mulu1.obj" of "Cmacro.lib" at 8000
locate : section5 in "div.obj" at 1000h linkafter section1
stack
locate : section6 in "div.obj" after section2
```

Note: One of **at** or **after** can be used with one of **linkat** or **linkafter**.

In all modes, address' number-bases are defaulted to hexadecimal, and can not be changed.

"linkat" or "linkafter" can be used for indirect linking.

Option "common" can be used for overlapped feature. Same section name of different obj file will be overlapped.

Option "reference" can be used for reference only feature. This section will be linking, but its content will not be output.

3.14 Automatic Mode

Automatic mode is a handy way to operate the linker. In this mode, only need to specify the input file(s) and library file(s) to be linked in a data file with extension of .ARY. You can specify the path from which to search for the library file(s). With this data file, the linker will link all the inputs automatically according to the section's default attribute except user specified in .Lik Data File. That is, linker will read these two config files (.ARY and .Lik) simultaneously. The .ARY file has following syntax.

3.14.1 Command Usage

xlink16 -a file1.ary file2 -body name [-bfile file3] [-rfile file4] [–novec] [–noitbl] [-nobdy] [-tsklen len][-initdata] [-iramnpage0]

-a:	Using automatic mode
file1: file2:	Required. You must specify .ary filename.
name:	Option. Specify output filename
file3: file4:	Specify body name.
novec:	Option. Specify body filename. (Default is body.dat)
noitbl:	Option. Specify external symbol file for reference. (Generated by Xlink16) Option.
nobdy:	Not generate interrupt vector table in executive file.
len:	Option. Not generate initial table(for IRAM/ISRAM section) in executive file. Option.
initdata:	Not check -body and -bfile match or not.
iramnpage0:	specify the size of .tsk file.
	Option. Specify that initial table can be placed at nonpage0.
	Option. Specify that IRAM can cross page and use new startup code

3.14.2 Keywords

Keywords are not case sensitive. They are described in the order in which they should appear.

Table 3-6

Keyword	Modifiers
Obj :	*Filename[, filename...]*
Lib :	*Filename[, filename...]*
Libpath :	*Pathname*
[options :]	tsk, map, adhighlevel, microtek, m37
Output :	*Filename*
PrjPath :	*Project Pathname*
SetLoadAdr	XXSymbol in "XXX.obj" or XXSymbol in "XXX.obj" of "XXXX.lib"

In this table, and in the following syntax definitions, the colon after a keyword is for clarity only and is optional.

Options

Function: Controls the creation and destination of a load map, and the formats of output symbol and code files.

Syntax: ***options: option[, option ...]***

Obj

Examples: `options: tsk, map`

Function: Names the input file(s) to be linked. The default object extension is **.obj,** but you can specify.

Syntax: ***obj : "filename"[," filename" ...]***

Examples:
```
obj : "mytest", "yourtest"
obj : "mytest.obj", "yourtest.obj"
```

Lib

Function: Names the library file(s). The default extension is **.lib.**

Syntax: ***lib : "libfile"[," libfile" ...]***

Examples:
```
lib : "Cmacro.lib", "Printf.lib"
```

Libpath

Function: Specify the path to be searched for the library file(s).

Syntax: ***Libpath : "pathname"***

Examples:
```
Libpath : "c:\ide\lib"
```

Output

Function: Names the output file. The default extension is the processor's default code file format extension, but you can specify otherwise.

Syntax: ***output : "filename"***

Examples:
```
output : "ourtest"
output :
"ourtest.ext"
```

SetLoadAdr

Function: locate the symbol of section to load Address. (Note: It is only available in Automatic Mode)

Syntax: **SetLoadAdr: XXSymbol in "XXX.obj" or SetLoadAdr: XXSymbol in "XXX.obj" of "XXXX.lib"**

Examples: SetLoadAdr: _CMOS_HW_Start in "CMOS_HW.obj" of "FPN.lib"

Note:

- In Automatic mode, the linker will link all the input(s) automatically and generate the link file and executable code file according to the command line option.
- The usage of options in automatic mode is same as that Enhanced data file mode.
- Map and .Sym files generation are default setting.

3.15 Offsets

In all modes, the linker requires an offset for each section of each file in a link. In automatic mode, if you omit an individual section's offset, the linker will link automatically based on body configuration loading the section at proper address.

3.16 Indirect Linking

When a program needs to modify data located in ROM by overwriting it, it cannot do so unless the data is moved into RAM. Indirect linking achieves this by stacking the data normally at a **load** address in ROM, but linking it to a **run**-time address in RAM.

3.17 Direct

Your program resides in ROM. The data consists entirely of lookup tables or constants, intended only to be read, and never to be modified. There is no reason to move it out of ROM.

3.18 InDirect

The same program resides in ROM, but has initialized data whose contents will change as the program runs. You need to move the data from its **load** address in ROM to a **run**-time address in RAM. Link indirectly. The table shows the differences in run-time link addresses.

Table 3-7

Direct		**Indirect**	
Load	**Run**	**Load**	**Run**
(ROM) CODE DATA	(ROM) CODE DATA	(ROM) CODE D ATA	(ROM) CODE
(RAM)	(RAM)	(RAM)	(RAM) DATA

For examples of indirect linking, see ***Appendix C***

3.19 Important rules for .Lik File

3.19.1 Enhanced Data File Mode / Automatic Mode

At least the first section in a link must be linked directly; the linker has a fixed address from which to calculate the indirect addresses.

Indirect sections are stacked in the same order as if they were direct.

You **can** indirectly link a section by using the keywords, **linkat** and **linkafter.**

If you try to link the same section twice, directly or indirectly, the linker will carry out the first link and will then refuse to perform the second link. An error message will refer you to the **second** attempt.

3.19.2 Lik File Syntax

Locate: section_name [in object [of library]] at address [linkat address / linkafter section] [common/reference]

Locate: section_name [in object [of library]] after section [linkat address / linkafter section] [common/reference]

3.20 Project Information Provide by Linker

The Linker links separated assembly programs, modules, files and sections into a single integrated program. It also provides project linking information for user program.

- __sn_sp_value: Used for SP register setting. It records the tail of free RAM at bank 0.
- __sn_ram_end: It records the maximum RAM address used by user project.
- __sn_ram_min: It records the minimum RAM address in specified body.
- __sn_ram_max: It records the maximum RAM address in specified body.
- __sn_init_table: It records the initial table address for startup code to initialize IRAM/ISRAM/IRAM_BANK0 and it can be assigned absolute address.

3.21 Symbol Table

3.21.1 2500AD High Level

L Linker Option:	**C**
Symbol Table Filename:	same as Linker output file
Filename Extension:	**.sym**

Table 3-8

Byte Sequence	Comment

Byte Sequence	Comment
FCh	ID byte
Version	2 bytes, MSB first
Time Stamp	4 bytes, LSB first (value in seconds)
Number of Object Modules	2 bytes MSB first
FDh	Start of Module
Object Module Name Size	1 byte
Object Module Name	
Object Module Type	0 = Assembly; 1 = C
Rest of Object Module Size	4 bytes MSB first
Symbol Address Size Code	1 byte See following table
Symbol Name Size	1 byte
Symbol Name	
Symbol Value	See Address Size Codes table
Flag Bytes	12 bytes
Symbol Size Code	1 byte See Address Size Codes table
LSB Filename #	The file in which this symbol was defined
MSB Filename #	
Indirect Count	1 byte
Structure Template	
Array Dimensions	
Spares	8 bytes reserved
The whole of this shaded area is repeated for each	symbol
. . . Rest of Symbols and Values . . .	
FDh	End of Module/Start of Module
Next Module Information (repeated as above)	
FDh	End of Last Module
FBh	Start of Filenames
Number of Filenames	2 bytes - MSB first

Byte Sequence	Comment
First Filename . . . Last Filename	Each Filename is Terminated by a 0 (zero)
FFh	End of File

Address Size Codes

Table 3-9

Code	Size	Byte Order	Code File Type
5	32-bit	MSB first	s37, Executable

3.21.2 Microtek

L Linker Option:	**M**
Symbol Table Filename:	same as Linker output file
Filename Extension:	**.sym**

Byte Sequence	Comment
FEH	Start of Module
Size of Module Name	
Module Name	
Rest of Module Length	3 bytes long
. . . Size of Symbol Address . . .	2 = 16 bits 3 = 24 bits 4 = 8086, 80186, 80286 5 = 32 bits
Size of Symbol	
Symbol Name	

Low Byte of Address	
High Byte of Address	
. . . Rest of Symbols and Values . . .	
FEH	End of Module
Next Module Information (same as above)	
FFH	End of File

3.22 Code File Format

3.22.1 Executable

L Linker Option:	**X**
Default Output File Extension:	**.tsk**

An executable code file is a pure binary file of opcodes and operands, with an assumed starting address of 0000H. Since the linker fills gaps between the end of each section and file, and the start address of the next with 0x0.

3.22.2 Motorola S37

L Linker Option:	3
Default Output File Extension:	.s37

Table 3-11

Field	Description
Record Type	Indicates the start of a record. It also identifies the record type as follows: ASCII S3 - Data Record ASCII S7 - EOF Record
Record Length	Specifies the record length that includes the Address, Data and Checksum fields. The 8 bit Record Length value is converted to two ASCII characters, high digit first.

Load Address	Eight ASCII characters, the result of converting the binary value of the address in which to begin loading this record. The order is: High digit of high byte of high word Low digit of high byte of high word High digit of low byte of high word Low digit of low byte of high word High digit of high byte of low word Low digit of high byte of low word High digit of low byte of low word Low digit of low byte of low word

Field	Description
	In an EOF record, this field has the program start address, or eight ASCII zeros.
Data	The actual data is converted to two ASCII characters, high digit first. There are no data bytes in the EOF record.
Checksum	The 8-bit binary sum of the record length, load address and data fields. The sum is then complemented (1's complement) and converted to two ASCII characters, high digit first.

3.23 Map File Format

3.23.1 Map Summary

The linker will create a map file (i.e. *filename* **.map),** if options 'D' is chosen for command line and prompt modes or options 'map' is chosen for enhanced data file mode. A map file can contain symbols defined in every file of the link. Cross references may also be included in the map file.

The SECTION SUMMARY shows all sections that are contained in the object files along with their load / run time addresses.

SECTION SUMMARY load / run time addresses will differ if linking a section indirectly.

The MEMORY SUMMARY shows all memory block status used by this project.

The linker can relocate listing files. After the linker relocates a listing file, symbol addresses/values are fixed, thus showing run time addresses/values.

Here is a typical map file followed by a description of all its elements.

```
Monday, January 14, 13:47:57, 2002


                Generalplus unSP Linker - Ver. 1.5.0 (Build:1)
                ------------------------------------------------------


Global Symbol Name   Global Value    Global Filename
*******************************************************************************
_RESET             8010       "C:\Program Files\unSP IDE\startupD.obj"
__sn_loop              802C         "C:\Program Files\unSP IDE\startupD.obj"
__sn_loop2             802B         "C:\Program Files\unSP IDE\startupD.obj"
```

```
__sn_init_table        8000       "Linker internal symbol"
__sn_sp_val          7FF        "Linker internal symbol"
_main               802D       "C:\Program Files\unSP
IDE\Ex\DTest3\Debug\DTest3.obj"
_RES_Table         0          "C:\Program Files\unSP
IDE\Ex\DTest3\Debug\Resource.obj"
*************************************************************************************

*************************************************************************************
*                        S E C T I O N    S U M M A R Y                          *
*************************************************************************************
* Section Name                            Startting Address   Ending Address    Size *
*************************************************************************************
*"C:\Program Files\unSP IDE\startupD.obj"
     *
*    DEBUG                                  ----               ----        ----       *
*    CODE                                   ----               ----        ----       *
*    DATA                                   ----               ----        ----       *
*    TEXT                                   ----               ----        ----       *
*    IRAM                                   ----               ----        ----       *
*    ISRAM                                  ----               ----        ----       *
*    RAM                                    ----               ----        ----       *
*    SRAM                                   ----               ----        ----       *
*    ORAM                                   ----               ----        ----       *
*    OSRAM                                  ----               ----        ----       *
*    NB_DATA                                ----               ----        ----       *
*    unSP_StartUp(TEXT)                    8010              802C      1D    *
*"C:\Program Files\unSP IDE\Example\DTest3\Debug\DTest3.obj"
     *
*    DEBUG                                  ----               ----        ----   *
*    CODE                                  802D              8051            25   *
*    DATA                                  8063              806D            B    *
*    TEXT                                  8009              800F            7    *
*    IRAM                                  8061 (37)           8062 (38)     2    *
*    ISRAM                                 805B (18)          8060 (1D)     6    *
*    RAM                                   28                36              F    *
*    SRAM                                   1                 17          17  *
*    ORAM                              1E                27          A   *
*    OSRAM                             0                 0           1       *
*    NB_DATA                           ----                ----          ----    *
*    MySec1(CODE)                      8052               805A        9 *
*    MyCode1(CODE)                    10000              1FFFF       10000 *
* "C:\Program Files\unSP IDE\Example\DTest3\Debug\Resource.obj"
         *
*    DEBUG                             ----              ----        ----       *
*    CODE                              ----              ----        ----       *
*    DATA                              ----              ----        ----       *
*    TEXT                              ----              ----        ----       *
*    IRAM                              ----              ----        ----       *
*    ISRAM                             ----              ----        ----       *
*    RAM                               ----              ----        ----       *
*    SRAM                              ----              ----        ----       *
*    ORAM                              ----              ----        ----       *
*    OSRAM                             ----              ----        ----       *
*    NB_DATA                           ----              ----        ----       *
* "Init Table"                             8000              8008        9   *
************************************************************************************


************************************************************************************
*                         M E M O R Y    S U M M A R Y                        *
************************************************************************************
*    Type                          Total             Using       Remain          *
************************************************************************************
* "SRam/OSRam/ISRam"                 40                1E          22              *
* "Ram"                            800                39          7C7             *
* "First Bank Rom"                    7C00              6E          7B92             *
* "Other Banks Rom"                  40000             10000       30000            *
************************************************************************************
```

Linker Output Filename : .\Debug\DTest3.S37

Disk Mapping Filename : .\Debug\DTest3.map

Symbol Table FileName : .\Debug\DTest3.sym

Format : Microtek

Linker Errors :	0
Output Format :	S37

This shows the label, "_main" assembled in Dtest3.obj. Its value is 0x802D.

This is a breakdown of where the sections are located at **run time.** The predefined CODE section is generated when the user has any code not located in a predefined user section or specifically calls the CODE section. The MySec1 and MyCode1 sections are user-defined sections. The description field is left blank.

(1) This indicates where sections actually reside in memory; their **load address.** This address may differ from run address if a section has been indirectly linked. The size indicates the length (in hex) of the respective section.

(2) Ex. IRAM section of Dtest3.obj, its load address is 0x8061, run address is 0x37, the section size is 2-word.

4 Lib Maker

4.1 General Description

A library is a file containing reusable object code program modules. When you link a program, the linker can access a library, search it for any modules that program refers, and selectively add these modules to the linker output. The lib maker lets you create libraries and modify them by adding, removing, or replacing modules.

4.1.1 System Resources

The software to manage the extended memory is part of the distribution executable files. Given enough disk space, a library file may be any size.

4.1.2 Default Filename Extensions

Table 4-1

.obj	Input file to the lib maker
.lib	Output file from the lib maker

4.2 Operating Instruction

4.2.1 Commands

You can run the librarian from the system **command line,** or by an **IDE** menu lib maker tools. The command description of command line is listed.

A command has a long form. Commands are:

ADD	Add modules to a library
DEL	Delete modules from a library
FIND	Find and select a specified module
LIST	List modules' global symbols
NEW	Create or load a different library
REP	Replace current library module with a new version

-RemoveDbg	Remove debug information in all module of a lib
-Extract Module	Exact a module from the lib as a single obj file

4.2.2 Command Line

You can add commands and arguments to the calling command. Command line operation uses the long command forms and supports multiple commands. You can apply several commands in a command line.

4.2.3 IDE Menu Lib Maker Tools

You can also operate the library by IDE menu lib maker tools. IDE menu lib maker tools will support a GUI (graphical user interface) to manage modules of library.

4.3 Calling the Lib Maker

To call the lib maker from the system command line, first make sure you are in the directory that contains the executable file **xlib16.exe.** If you type:

xlib16<cr>

The main screen will display as follows, telling you the usage and some examples of xlib16.exe:

Generalplus unSP Lib – Ver. Release 1.4.0 (Build:0)

Usage: **XLib16 LibFileName Command [Argument Command [Argument ...]]**

Example: XLib16 MyLib.lib New

Example: XLib16 MyLib.lib Add MyModule

Example: XLib16 MyLib.lib Del MyModule

Example: XLib16 MyLib.lib Find MyModule

Example: XLib16 MyLib.lib Rep MyModule

Example: XLib16 MyLib.lib List

Example: XLib16 MyLib.lib New Add MyModu1 Add MyModu2 Add MyModu3 List

Example: XLib16 MyLib.lib List Add MyModule List

4.4 Command Line Operation

You can run the librarian, create or call a library and manipulate it directly from the system command line. The operating system dictates the maximum length of the command line.

Typing a **<cr>** terminates and executes a command line.

For example:

```
Xlib16 test.lib new

; Create a new library whose name is test.lib

Xlib16 test.lib add modul1

; Add a module 'modul1' to the library 'test.lib'

Xlib16 test.lib del modul1

; Delete a module 'modul1' from the library 'test.lib'

Xlib16 test.lib find modul1

; Find a module 'modul1' in library 'test.lib'

Xlib16 test.lib rep modul1

; Replace the module 'modul1' in library 'test.lib' with an external module with the same

; name 'modul1'. If there is no 'modul1' in library 'test.lib', this operation will be failure.

Xlib16 test.lib add modul1 del modul2 rep modul3 list

; To the library 'test.lib', first add the module 'modul1', and then delete the module

; 'modul2'. After that, replace the modul3 with an extern module with the same name

; 'modul3'. Finally list all modules' global symbol.
```

* If the library name is correct, you can execute the 'new', 'list' and 'find' command successfully. For 'add', 'del' and 'rep' command, if one of these commands can not execute successfully, the command line including these commands can not execute.

4.5 Creating a module

There are three stages to the creation of a C library module.

- Create a library function C source file.
- Compile the C source file.
- Assemble the compiled file

To create an assembly language library module, simply write the module in assembly language and assemble it.

4.5.1 Create C Source Files

Use an ASCII text editor to write a C source file for each function that you want to add to a library. For example:

```
/* filename = test1.c */
int test_add_int(int a, int b)
{
   return( a + b );
}
/* filename = test2.c */
float test_add_float(float a,float b)
{
   return( a + b );
}
```

4.5.2 Compile C Source Files

Compile these files with GENERALPLUS C Compiler by entering:

GCC -S -test1.c -o test1.asm

GCC -S -test2.c -o test2.asm

The **-S** switch tells the compiler to generate assembly language.

4.5.3 Assemble the compiled files

If you want to assemble the assembler source file and add the object file as a module to a library, just assemble each **.asm** file separately with the **XASM16** Assembler by entering:

XASM16 test1.asm

XASM16 test2.asm

The output files will be **test1.obj** and **test2.obj.**

To add those two new routines to a library, call the librarian, load the library and use the ADD command.

4.5.4 Library Routine

Since the C compiler always prefixes an underscore (_) to any label that it generates, a C program call to an assembly language routine expects the routine name to begin with an underscore (_). Therefore the name of any library routine that you create in assembly language should start with an underscore (_). The names of all library routines and modules start with one or more underscores (_).

When calling a library routine or a function, there are several rules need to be followed:

a. Arguments passing

Arguments are pushed on the stack in reverse order (right to left). If necessary, all arguments are converted to their corresponding data type declared in the function call prototype. However, if the function call occurs before the function declaration, the arguments will be passed to the called function without any conversion.

b. Stack maintenance

It is the caller's responsibility to pop the arguments from stack.

c. Value returning

Returned vales are stored in the Register R1 for 16 bits and in register pair for 32 bits with low word in R1 and high word in R2. Structures are returned in the Register R1 as pointers to the structures.

d. Register saving

The compiler generates prolog and epilog to save and restore the PC, SR and BP registers. PC and SR are auto pushed into stack by "CALL" instruction. PC and SR are auto popped from stack by "RETF" or "RETI" instruction.

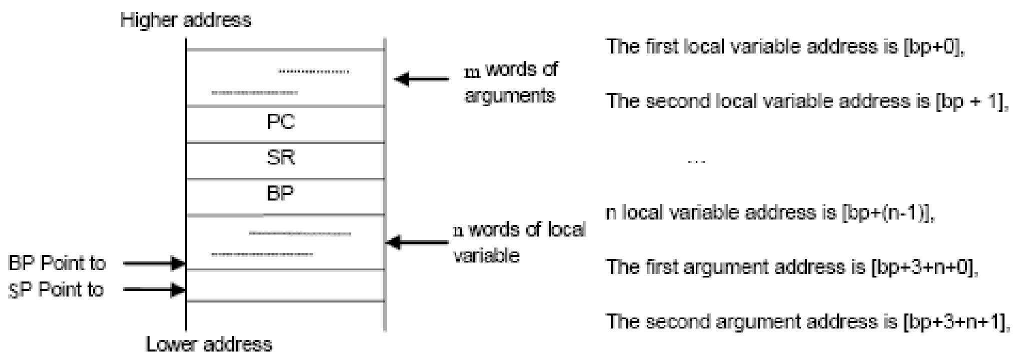
Higher address
m words of arguments
PC
SR
BP
n words of local variable
BP Point to
SP Point to
Lower address
The first local variable address is [bp+0],
The second local variable address is [bp + 1],
...
n local variable address is [bp+(n-1)],
The first argument address is [bp+3+n+0],
The second argument address is [bp+3+n+1],

Figure 4-1

e. Stack scheme

Assume each local variable/argument size is one word

1st local variable address is [bp++0]

2nd local variable address is [bp + 1]

nth local variable address is [bp + (n-1)]

1st argument address is [bp + 3 + n + 0]

2nd argument address is [bp + 3 + n + 1]

mth argument address is [bp + 3 + n + (m-1)]

f. Pointer

Pointer is implemented by 16-bit in this compiler. Function pointer does not really point to the entry of a function, instead it points to an address in __function_entry section where the real function start address is placed into two continue words.

Example1:

```
.TEXT
__function_entry: .dw seg _function, offset _function
long test_long(int a,long b)
{
return( a + b );
}
public _test_long
.CODE
_test_long: .proc
    push bp to [sp];                 // Store bp value
    bp = sp + 1;
    r4 = [bp + 0 + (3)];          // Get first parameter “a” from stack
    r3 = r4;                        // Assign a to r3
    r4 = 0;
    r3 = r3;                      // Judge parameter a is a negative or
    not
    jpl1;                         // If positive, skip next instruction
    r4 -= 1;                      // If negative, extend the sign bit to
    r4
    r1 = r3;                      // The low word of "a" add low word of
    "b"
    r2 = r4;
    r1 + = [bp + 0 + 4];
    r2 + = [bp + 0 + 5], Carry; // The high word of "a" add high word of
    "b"
    Carry;                      // with carry the return value is stored in
    r1
                              // and r2 the low word in r1, the high word in
                              r2
L1: pop bp from [sp];           // Restore bp value
    retf;
     .endp;
int test_int (int a,int b)
{
     return( a + b );
}
public _test_int
_test_int:  .proc
    push bp to [sp];
    bp = sp + 1;
    r4 = [bp + 0 + 3];          // Get first parameter "a" from stack
    r4 + = [bp + 0 + 4];         // Add "a" with second parameter "b"
    r1 = r4;                    // Store the addition at r1 as result
L2: pop bp from [sp];
    retf;
   .endp

void main(void)
{
   int i,j;
   long l;
   i = 2;
   j = 3;
   l = i;
   i = test_int(i,j);
   l = test_long(i,l);
   return;
}
public _main
_main:    .proc
       push bp to [sp];
       sp = sp-4;          //Alloc space; int is one word, long is two word
       bp = sp + 1;
       r4 = 0x2;
       [bp + 4 + (-1)] = r4;    // Assign 2 to I
       r4 = 0x3;
       [bp + 4+ (-4)] = r4;     // Assign 3 to j
       r4 = [bp + 4 + (-1)];    // Get i value
       r2 = 0;
       r1 = r4;                 // Assign i to the low word of l
       jpl1;                    // If i is positive, skip the next
       instruction


       r2- = 1;                 // If i is negative, extend the sign bit
       //[bp + 4 + (-3)] = r21; // Assign the value of l lower word
       [bp + 1] = r1;           // Assign the value of l higher word
       [bp +2] = r2;            // Get j
       r3 = [bp + 4 + (-4)];    // Push j as parameter 2
       push r3 to [sp];             // Push I as parameter 1
       push r4 to [sp];
       call _test_int;          // Discard parameter
       sp = sp + 2;             // Get the return value
       r4 = r1;                 // Assign the return value to I
       [bp + 4 + (-1)] = r4;    // Get the value of variable l
       r3 = [bp + 1];
       r4 = [bp + 2];           // Push parameter l to stack
       push r3, r4 to [sp];     // Get the value of variable a
       r4 = [bp + 4 + (-1)];    // Push parameter a to stack
       push r4 to [sp];
       call _test_long;         // Push parameter a to stack
       sp = sp + 3;             // Assign the return value to l
       [bp + 1] = r1;
       [bp + 2] = r2;
L3:
       sp = sp + 4;             // Release space
       pop bp from [sp];        // Restore bp value
       retf;
       .endp
       .end                    // End asm code
```

5 Appendix A. Error Message

5.1 Assembler Errors

This list includes the error messages for XASM 16 Assemblers.

A0000: Syntax error

This occurs when an instruction or expression is not fit to the XASM16's format.

A0001: '...' already defined

A symbol '...' that has already been defined couldn't be redefined.

A0002: Bad use of local symbol '...'

Local symbol cannot be used as macro name, procedure name, section name, struct name, struct variable name and constant. An error will occur if a local symbol is used as one of those.

A0003: Bad use of keyword '...'

This occurs when an instruction or a directive is used as a symbol.

A0004: '...' already defined

A local symbol with same name has already defined.

A0005: Local symbol '...' can not be declared as external

A local symbol is valid only within its "local area" and cannot be declared as external.

A0006: '...' Illegal forward reference or symbol undefined

This occurs when a variable is referenced but has not been defined yet.

A0007: '...' undefined

The symbol that you referenced has not been defined yet.

A0008: Local symbol '...' cannot be declared as public

A local symbol is valid only within its "local area" and cannot be declared as public.

A0009: '...' cannot be declared as public---wrong type

The symbol is defined as number constant and cannot be declared as public.

A0010: '...' cannot be declared as public because it's defined with VAR

The symbol that defined with keyword VAR cannot be declared as public.

A001 1: '...' should represent a number

The parameter should be a number in macro reference.

A0020: File '...' can not be opened

The file specified could not be accessed or opened (misspelled?).

A0021: Include inside a macro

The file include operation cannot be used inside the definition of a macro.

A0030: Bad use of macro name '...'

This occurs when the use of macro name is not fit to the XASM16's format.

A0031: Cannot define a macro inside another macro

You attempt to define a macro inside the definition of another macro.

A0032: IFMA used outside macro

The directive IFMA must be used with directive MACRO and ENDM

and cannot be used outside macro definition.

A0033: Cannot define a macro inside the definition of a structure

You attempt to define a macro inside the definition of structure.

A0034: Bad declaration of dummy parameters

The declaration of dummy parameters in macro definition isn't fit to the XASM16's format.

A0035: Bad representation of real parameters

The representation of real parameters doesn't math with the declaration of dummy

parameters.

A0036: ENDM expected before end of file

In the source code file, the numbers of MACRO and ENDM are not balance, and need an ENDM to keep the balance.

A0050: Bad use of section name '...'

This occurs when the use of section name is not fit to the XASM16's format.

A0051: Cannot define more than ... sections

The maximal section number in XASM16 is 4096. You cannot exceed this limitation.

A0052: Instruction or data emitted with no section

All instruction and data must be used within some section.

A0053: Symbol defined with no section

A symbol must be defined within some section.

A0060: Number overflow

The number is too large and overflows.

A0061: Cannot add two address expressions

In XASM16, cannot add two address expressions together.

A0062: Cannot subtract two address expressions that belong to different sections

In XASM16, subtract two address expressions that belong to different sections is illegal, but subtract two address expressions which belong to same sections is legal.

A0063: Cannot subtract a number expression with an address expression

In XASM16, subtract a number expression with an address expression is illegal.

A0064: Operator ('!', '%', '&', '*', '-', '/', '^', '|', '~', '<<', '>>'), number expression expected

The operator symbol listed above must be operated on number expression.

A0065: Divided by zero

The divisor operand has evaluated to 0.

A0066: Bad use of external symbol

The external symbol used in this way is illegal.

A0067: Operator ('%', '&', '^', '|', '~', '<<', '>>'), illegal operand type of float

The operator symbol listed above cannot operate on float type directly.

A0080: Illegal conditional assembly

The format of conditional assembly is incorrect.

A0090: Missing field name in structure definition

Missing field name in structure definition.

A0091: Field name '...' already defined

The field with same name in structure definition has already been defined.

A0092: '...' not a valid field name

The field name you referenced is not defined during the definition of structure.

A0093: '...' is not a structure variable

The symbol you referenced is not a structure variable.

A0094: Cannot define a structure variable in another structure definition

In XASM16, that define a structure variable in another structure definition is illegal.

A0095: Bad use of structure name '...'

This occurs when the use of structure name is not fit to the XASM16's format.

A0096: Cannot nest a structure definition inside another structure definition

In XASM16, nest a structure definition inside another structure definition isn't allowed.

A0097: Cannot emit instruction into a structure definition

During the definition of a structure, you cannot emit an instruction.

A0098: Define a field but not yet allocate memory for it

During the definition of a structure, you define a field but not specify the data type for it.

A0099: ENDS expected before end of file

In the source code file, the numbers of MACRO and ENDM are not balance, and need ENDS to keep the balance.

A0100: Field ... of ... cannot be re-initialized

In XASM16, only those fields, which reserve memory without the use of comma ',', can be initialized by initialization list.

A0101: Field ... of ... is re-initialized unsuccessfully

In XASM16, only those fields, which reserve memory without the use of comma ',', can be initialized by initialization list.

A0102: Field ... of ... cannot be re-initialized with a string

In XASM16, some field of a structure cannot be re-initialized with a string.

A0103: The string is too long, and the field '...' of '...' is re-initialized unsuccessfully

The size of field that defined in structure is less than that referenced in initialization list.

A0104: Cannot define or switch a section inside the definition of a structure

Inside the definition of a structure, define or switch a section is not allowed.

A0105: Include inside a structure definition

Cannot include a file inside a structure definition.

A0107: Cannot use VAR to allocate memory in a structure definition

Cannot use VAR to allocate memory in a structure definition.

A0110: Cannot define a string with FLOAT

Cannot define a string with directive FLOAT.

A0111: Left operand of DUP should not be negative

The number in the left of DUP represents the repeat number and should not be negative.

A0112: '...' operand type doesn't match the directive of storage allocation

The operand is too large and doesn't match the directive of storage allocation.

A0113: DB cannot be used

In XASM16, the use of operator type DB isn't allowed, because we cannot store a value in 8-bit location.

A0114: Bad use of Imm6

This occurs when modifier Imm6 is used to modify a label or external symbol.

A0115: Bad use of A6

This occurs when modifier A6 is used to modify a number expression.

A0116: Bad use of OFFSET

This occurs when modifier OFFSET is used to modify a number expression.

A0117: Bad use of SEG

This occurs when modifier SEG is used to modify a number expression.

A0118: Bad use of HIGH6

This occurs when modifier HIGH6 is used to modify a number expression.

A0120: Cannot nest a procedure definition in a structure definition

In XASM16, cannot nest a procedure definition in a structure definition.

A0121: Expect to switch back to the same section before ENDP

A procedure must be in the same section. Cannot switch section during the definition of procedure.

A0130: SEG cannot modify number expression

The modifier SEG cannot modify number expression.

A0131: OFFSET cannot modify number expression

The modifier OFFSET cannot modify number expression.

A0132: A6 cannot modify number expression

The modifier A6 cannot modify number expression.

A0133: HIGH6 cannot modify number expression

The modifier HIGH6 cannot modify number expression.

A0140: Address expected

The function call instruction must call a function name and the function name represents the address of function.

A0141: Invalid loop counter, integer 1 to 16 expected

In sum of registers multiplication instruction, the loop counter must in the range of 1~16.

A0142: BP or R5 register expected

In the instruction with indirect memory access (BP and 6-bit immediate), the base pointer must be BP or R5.

A0143: Source register and destination register should be the same

The source register and the destination register should be the same.

A0144: Invalid BP offset, integer 0 to 63 expected

In the instruction with indirect memory access (BP and 6-bit immediate), the valid offset should be in the range of 0~63.

A0145: Cannot jump to external label

In XASM16, the jump instruction cannot jump to external label.

A0146: Cannot jump to other section

In XASM16, the jump instruction cannot jump to the label that lies in other section.

A0147: Jump too far

The jump range is too large and it is limited to PC 63.

A0148: Float unexpected

The float number in the instruction is unexpected.

A0149: Cannot use Imm6 mode cause value too large

Cannot use Imm6 mode.

A0150: Cannot pop into SP

In XASM16, cannot pop into SP.

A0151: Shift counter should be non-negative integer 1 to 4

In the instruction with shift (Register), the shift counter should be in the range 1~4.

A0152: Cannot use A6 mode

The operator exceeds the range 0~63 and cannot use A6 mode.

A0153: Source registers cannot be same as destination register

In the Registers Multiplication (Mul) instruction, the source register cannot be same as the destination register and the source register cannot be R3 or R4.

A0154: Address expected

The Goto instruction should jump to an address.

A0155: Source registers cannot be same as destination register

In Sum of Registers Multiplication (Muls) instruction, the source register cannot be same as the destination register.

A0156: Register PC cannot be used in this addressing mode.

This occurs when destination/source register is equal to PC.

A0157: Register SR cannot be used in this addressing mode.

This occurs when destination/source register is equal to SR.

A0158: Register SP cannot be used in this addressing mode.

This occurs when destination/source register is equal to SP.

A0159: Invalid bit operation offset, integer 0 to 15 expected

In bit operation instruction, the offset must in the range of 0~15.

A0160: Source register R4 or R3 was expected in 32-bit shift operation.

In the instruction with 32-bit shift operation, only R4 or R3 can be used.

A0161: Destination register R2 was expected in exp instruction

In the instruction with exp operation, only R2 can be destination register.

A0162: Source register R4 was expected in exp instruction

In the instruction with exp operation, only R4 can be source register.

A0163: Divisor register R2 was expected in div instruction

In the instruction with div operation, only R2 can be divisor.

A0164: Cannot push over 7 registers in single operation.

Max register number is 7 for a push operation.

A0165: Cannot push SP into stack/memory.

No support this instruction type for *unSP* 1.0 and *unSP* ® 1.1.

A0166: Source register can't been SR register for this instruction

A0167: '...' isn't a compile time variable(declared by EQU/VDEF/DEFINE)

A0168: IRAM,ISRAM section can't specify address.

A0169: this chip no support such instruction type

A0170: Size of section '...' exceeds 64K

5.2 Linker Errors

These are the most commonly encountered linker errors.

L0000: MFC initialization failed

The Xlink16 cannot run in current platform. An error occurred while MFC library was being

initialized.

L0010: Cannot open the file ...

The file specified could not be accessed or opened (misspelled?).

L0011: Link file ... is empty

The link file is empty and contains no relocation information, maybe it is corrupted.

L0012: Cannot open the file ..., return from Make_Exe proc

While generating code for the executable file, the Xlink16 cannot open the executable file.

L0013: Cannot open the file ..., return from making the symbol file

While making the symbol table file, the Xlink16 cannot open the executable file for writing code.

L0014: File destroyed: ...

The file may be corrupted.

L0020: Cannot locate … section automatically, please manually locate it

This occurs when the sections are too much to be located by Xlink16 automatically.

L0023: Cannot locate ... section at ram address 0-63

The sections variables are too much to be located at ram address 0-63.

L0031: The ram variables are too much to be located

The ram variables are too much to be located at ram address.

L0032: Unknown linker option...

You input an option word that the Xlink16 cannot identify.

L0040: Cannot find this body information in body file

Cannot find user specified body in body file

L0041: No interrupt vector information in body file

In order to locate the interrupt vector correctly for Xlink16, the file body.dat must contain

interrupt vector address information.

L0042: Page 0 hasn't enough space for init table

There are too much code to be located in page 0.

L0043: … function hasn't been defined.

The … interrupt service routine hasn't been defined.

L0050: No object filename

This occurs when you don't input the object file to be linked in ARY file or link file.

L0051: Unwanted address ...

This occurs when you declare a section's address that has not been used for command line mode.

L0052: Illegal address ...

This occurs when you input section's address in link file with a wrong format that the Xlink16 cannot identify.

L0053: Syntax error

This occurs when your input doesn't fit to the format of ARY file or link file.

L0054: Not point out object name for defined section

In order to rename a section, you should specify the object file where the section lies.

L0055: No redefined section name

In order to rename a section, you should specify the renamed name.

L0056: No address after the word "At"

In the location of section, there must be an address after the word "At".

L0057: No section name after the word "After" in link file

In the location of section, there must be a section name after the word "After".

L0058: No address after the word "LinkAt"

In the location of section, there must be an address after the word "LinkAt".

L0059: No section name after the word "Linkafter"

In the location of section, there must be a section name after the word “Linkafter”.

L0060: The section ... has not been located

In the enhanced data file mode, a section must be located. Otherwise it will be located at address 0x00.

L0061: Define two times ...

The public symbol was redefined in different object or library.

L0062: The section ... has not been defined at any obj file

In enhanced data file mode, you referenced a section that has not been defined at any file.

L0063: Cannot locate ... section at addr ...

This addr has been used by other section or no suitable block for allocation.

L0064: The addr space of section ... and ... is overlapped

These two sections has conflicting address space.

L0065: Text section is too big to fill in first bank

There is no suitable size block for allocation.

L0066: This address ... has been used by other section

This address has been used twice. The address is overlapped by user.

L0067: No address after the character “with”

There must be an address after the word "with" in linking script.

L0068: Illegal alignment…, the value must be positive.

Alignment value must be positive.

L0069: The section … has an ADDR directive.

The order for this section would be cancelled in linking script.

L0070: The section ... has not been located before locate the section ... after it

In the enhanced data file mode, a section must have been located before being used to locate other section.

L0071: The section ... has not been located before locate the section ... linkafter it

In the enhanced data file mode, a section must have been located before being used to locate other section.

L0080: The external symbol ... has not a public definition

You referenced a local symbol defined in other file as an external symbol. The solution is to declare this symbol with public attribute in the file that has been defined.

L0100: ... end address is little than start address in BODY file

This occurs when the end address of RAM, ROM or I/O port is little than start address in BODY file.

L0111: Obj type is not Generalplus

The specified file is not Generalplus object file. . The file may be corrupted

L0112: Obj file has been destroyed. Return from make exe file

The Xlink16 detects that the object file has been destroyed and cannot make executable file

L0113: This project need init table for IRAM/ISRAM section..

If there are a few variable in IRAM/ISRAM section. User cannot specify "-noitbl" parameter for Xlink16.

L0115: Section ... is older obj file format. Please update resource compiler.

L0116: Section … in … cross different bank.

L0117: No content in obj file. There is no instruction written in obj file.

L0120: The library file ... is not Generalplus library

The specified file is not Generalplus library file. . The file may be corrupted

5.3 Lib Maker Errors

These are the most commonly encountered librarian errors.

L0500: MFC initialization failed

The Xlib16 cannot run in current platform. An error occurred while MFC library was being initialized.

L0510: The second argument must be a lib file

The second argument must be a GENERALPLUS library file, can not be other library file and the library file's extension must be '.lib'.

L0511: No FIND argument

In the command line, command FIND has no argument.

L0512: No ADD argument

In the command line, command ADD has no argument.

L0513: No REP argument

In the command line, command REP has no argument.

L0514: No DEL argument

In the command line, command DEL has no argument.

L0515: Cannot identify the command.

In the command line, an unknown command is used.

L0520: Read Lib Error

An error is occurred when read the specified library.

L0521: Write Lib Error

An error is occurred when write the specified library.

L0522: Cannot find the module.

Cannot find the specified module in the library file.

L0530: Cannot open the file.

The file specified could not be accessed or opened (misspelled?).

L0531: File destroyed:

Maybe the file is corrupted.

L0532: The bTag is valid. The Obj file could has been destroyed

Perhaps that the object files or library files to be linked have been destroyed.

L0540: The file ... is not Generalplus lib

The specified file is not GENERALPLUS library file, maybe it is another type library file or not a library file.

L0541: Obj Type is not Generalplus

The specified file is not GENERALPLUS object file, maybe it is another type object file or not an object file.

L0542: "…" is already exist

L0543: The Module "…" is already exist

L0544: The public symbol "…" is already exist in the Module

6 Appendix B. Additional Assembler Info.

6.1 Assembly Time Operators

Calculation operators are grouped in the following table.

All calculations use 16-bit integer arithmetic.

All comparisons return **1** for true and **0** for false.

Table 6-1

Calculations	
Operator	**Description**
&& ! ||	Logical AND Logical NOT Logical OR
& or AND ~ or NOT | or OR	Bit AND Bit NOT Bit OR
+ -	(Unary) optionally specifies a positive operand (Unary) negates the following expression.
* /	Unsigned multiplication Unsigned division
+ -	Addition Subtraction
>> <<	Shift the preceding expression right with 0 fill. The subsequent expression gives the number of shifts. Shift the preceding expression left with 0 fill. The subsequent expression gives the number of shifts.
The following operators must start and end with a space, or a tab. The table shows the periods. (Each processor has only those operators relevant to its address sizes.)	
IM6	(Unary) keeps bits 0 – 5 of a number expression.
A6	(Unary) keeps bits 0 – 5 of an address expression. (Necessary for re-locatable address values).
A16	(Unary) keeps bits 0 – 15 of an address expression.

Calculations	
Operator	**Description**
	(Necessary for re-locatable address values).
SEG	Keeps bit 16 – 21 of a 32-bit address expression and shift it to a 6-bit address. (Necessary for re-locatable address values).
SEG16	Keeps bit 16 – 31 of a 32-bit address expression and shift it to a 16-bit address. (Necessary for re-locatable address values).
OFFSET	Keeps bits 0–15 of a 32-bit address expression. (Necessary for re-locatable address values).
HIGH6	Keeps bit 16 – 21 of a 32-bit address expression and shift it to a 16 bits address. (The positions are the same as SR:DS) (Necessary for re-locatable address values).
BYTE_SEG	Applied for *unSP*-1 .3. Multiply a 32-bit address expression by 2, keeps bit 16 – 31 as a 16-bit address. Used to calculate the segment value of an address in byte domain. (This is necessary for re-locatable address values).
BYTE_OFFSET	Applied for *unSP-1.3.* Multiply a 32-bit address expression by 2 and keeps bit 15 – 0 as a 16-bit address. Used to calculate the offset of an address in byte domain. (This is necessary for re-locatable address values).
Comparisons	
==	Equal
!=	Not equal
>	Greater than
<	Less than
>=	Greater equal
<=	Less equal

The priorities of the calculation operators are shown in ascending order as following. The operators in the same line have a same priority. You can force different priorities by using parentheses.

'&&', '||'

'|' or 'or', '&' or 'and', '^'

'==', '!='

'>=', '>', '<=', '<'

'<<', '>>'

'+', '-'; Addition, Subtraction

'*', '/', '%'

'+', '-'; (Unary) specifies a positive or negative operand.

'! ', '~' or 'not'

6.2 Operations on Externals

- Although you may declare multiple externals (see ***Assembler-EXTERNAL),*** the assembler does not support addition, subtraction, or, and, or any logical expression involving more than a single external. This declaration is valid:

  .EXTERNAL CONT1,CONT2,CONT3

  This instruction is not valid:

  R1 + = (CONT1 & CONT2 & CONT3)

- Declare an external PAGE0 variable:

  In Prog1.asm

```
.SRAM

.PUBLIC
sym1 .VA
R sym1
…
```

  In Prog2.asm

```
.EXTERNAL sym1:A6

…

R1 = [ sym1];

// Cause sym1 will be at address 0-63, this instruction will be A6
instruction type
```

6.3 Spaces

In XASM16, the relation operator, bit operator, modifier operator and logical operator between two symbols must be separated by spaces.

Exp1 > Exp2

Exp1 && Exp2

SEG label1

6.4 Pre-defined Sections

The XASM16 has ten pre-defined sections: **CODE,** NB_DATA, **DATA, TEXT, CTOR, DTOR, IM, NB_MERGE, LINKONCE, VTBL, ORAM, OSRAM, RAM, IRAM, SRAM , ISRAM, RAM_BAN K0,IRAM_BAN K0,ORAM_BAN K0.**

In general, basic naming rules for ram:

I: It will be initialized by startup code

O: Overlay

S: Only be assigned RAM address 0~63 , apply for A6 addressing mode

The priorities of linking are shown in ascending order as following.

ROM:

DATA

N B_DATA

LINKONCE

CODE

NB_MERGE

VBTL

TEXT

IM

DTOR

CTOR

RAM:

IRAM

RAM

ORAM

ISRAM

SRAM

OSRAM

RAM_BAN K0

I RAM_BAN K0

ORAM_BAN K0

6.5 User-Defined Sections

You can generate your own section names with the SECTION directive. Each section name can be up to 32 characters long. You can have up to 4096 user-defined sections. In XASM16, not support you to control from your source file the way in which the section is linked. If you want to control the way in which the section is linked, you can specify the options in linker script.

6.6 Section Directive

6.6.1 SECTION

Group:	Assembly Mode
Function:	Create a user-defined section
Syntax:	***label: .SECTION .attribute [,.ADDR = value]***
Note:	The attribute can be one of the ten predefined section-names. The user-defined section has the same link attribute with its attribute. After you define a section,

	you can switch to and from it using the name as a mnemonic. When using the section name in this way, you can precede it, as you can all directives, with a decimal point.
Example:	See code below

```
.CODE                          //Set the pre-defined CODE section
Set_Freq:     R1 = 0x7010;
         [R1] = R2;
         .DATA             // Switch to pre-defined DATA
section
Key_Table:    .DW 0x20E2, 0x1A7B, 0x3167
// This byte goes into the DATA section
section1:     .SECTION .CODE
; Define a new section. Defining it makes
; the section automatically active
         r3 = r3-0x10; // This instruction goes into section1
         .CODE        // Switch back to the CODE section
         r1 = 1;
         // This instruction goes into the CODE section
         .section1    // Switch to user-defined section1
         r1 = r1 + 2;
         // This instruction goes into section1
St_Table:     .DW 0X30
// Any user-defined section may contain code or data or both
         .TEXT        // Switch to predefined TEXT section
```

6.6.2 Section Summary

The linker generates a section summary table as part of its load map file. This contains in condensed form the contents of the load map. The table lists the names of all the sections in the link, in alphabetical order.

For each section, the table contains section name, start address, end addresses and section size that you actually use.

The start address is the address at which the linker relocates the section.

See ***Linker-Operating Instructions*** for a description of the Linker's handling of section names and of Indirect and Stacked linking.

6.7 Macros

6.7.1 Defining a Macro

A macro is a sequence of source lines to be substituted for a single source line. You must define a macro before you can use it. On pass 1, the assembler stores the definition and, when it reaches the macro name, substitutes the defined source lines. A macro definition may include arguments, substituted into any field except the comment field. Dummy arguments may not contain spaces. The start of a macro must be defined by a MACRO directive. The macro's name goes in the label field. The macro must end with an ENDM directive.

6.7.2 Calling a Macro

In a macro call, arguments may be of any type: direct, indirect, character string or register. Only an ASCII string bracketed by apostrophes may include spaces. (Apostrophes in the string must appear as double apostrophes.) So long as the dummy argument names are identical, arguments can be passed through to nested macros. Memory space is the only limit on macro nesting. Arguments must be separated by commas. Leading spaces and tabs are ignored. A single comma acts as a placeholder for a missing argument.

6.7.3 Argument Separators

In a macro body, valid argument separators are common ','.

Expression with space must be quoted by parenthesis (,).

For example: MacroName R1, (9 + 8 / 2)

6.7.4 Labels

Macro definitions can contain explicit (user-defined) or implicit (auto-defined) labels. The assembler will not alter an **explicit** label. Adding a # suffix to a label makes it **implicit,** and tells the assembler to substitute automatically a digital following an underbar (_) and 4-digit expansion number for the #. The label and its expansion may not exceed 32 characters.

```
instruction:          .MACRO    arg,val
                         arg
lab#:                 .DW       val;
                       .ENDM
; Calling the macro:   Instruction  nop,7
; Produces the following result:
                       nop;
lab_1_6416:           .DW       7;
```

6.7.5 String Concatenation

The character '@'(hex 40) is the string concatenation operator. You can concatenate onlyinside a macro.

6.7.6 Value Concatenation

The character '|'(hex 7C) is the value concatenation operator. You can concatenate only inside a macro. It is used to concatenate a string and an expression value enclosed in angle-brackets.

```
concat          .MACRO          arg

                mac_value:      .VAR      mac_value + 1

                arg|<mac_value>  .EQU     27

                .ENDM
```

The following macro call

```
mac_value       .VAR            0
                Concat          label
```

Will expand as

```
label1:         .EQU            27
```

6.8 Macros Examples

6.8.1 Number Comparisons

```
cmp_number: .MACRO arg1
.IFMA 0
.MACEXIT
.ENDIF
.IF 1==arg1
month:      .DW 1;
.MACEXIT
.ENDIF
.IF 2==arg1
month:      .DW 2;
.MACEXIT
.ENDIF
.IF 3==arg1
month:      .DW 3;
.MACEXIT
.ENDIF
```

```
.IF 4==arg1
month:       .DW 4;
.MACEXIT
.ENDIF
.IF 5==arg1
month:       .DW 5;
.MACEXIT
.ENDIF
.IF 6==arg1
month:       .DW 6
.MACEXIT
.ENDIF
.ENDM
```

6.8.2 Passing a label name into program code

```
store_label: .MACRO arg1
.DW ”arg1”;
.ENDM
; Call the macro as follows:
store_label abc
; The expanded macro reads as follows:
.DW ”abc”;
```

6.8.3 Argument Substitution in an operand field

```
employee_info1:   .MACRO    arg1, arg2, arg3
name:             .DW       “arg1”;
department:       .DW       “arg2”;
date_hired:       .DD       arg3;
.ENDM
; Call the macro as follows:
employee_info1    "John Doe", personnel, 101085
name:                 .DW       "John Doe";
department:           .DW       “personnel”;
date_hired:           .DD       101085;
```

6.8.4 Passing an argument into the label field

```
employee_info2:         .MACRO    arg1,  arg2,  arg3
arg1  :                 .DW       30h;
arg2  :                 .DW       10h;
arg3  :                 .DD       1999;
.ENDM
```

```
;Call the macro as follows:
```

```
employee_info2     name, department,
date_hired ; The expanded macro reads as
follows:
name:              .DW         30h;
department:        .DW         10h;
date_hired:        .DD       1999;
```

6.8.5 Recursion

In this recursive macro, **arg1** (count) controls the number of recursions. During each recursion the macro reserves four data word and fills them with the values specified by **arg2, arg3, arg4** and **arg5.** Each successful execution decrements the count.

```
reserve:          .MACRO              arg1, arg2, arg3, arg4, arg5

count:            .VDEF               arg1;

                 .IF    count==0

                 .MACEXIT

                 .ENDIF

count:            .VDEF               count-1

                 .DW                 arg2,arg3,arg4,arg5;

                 reserve
                                     count,arg2,arg3,ar
                 g4,arg5 .ENDM
; Calling the macro:
                 reserve             5,0x1,0x2,0x3,0x4

 ; Produces the result:
 count:          .VAR                5;
                 .IF                 count==0
                 .MACEXIT
                 .ENDIF
 count:          .VAR                count-1;
                 .DW                 0x1,0x2,0x3,0x4
                 reserve             count,0x1,0x2,0x3,0x4
                 . . .
 count:          .VDEF               count;
                 .IF                 count==0
                 .MACEXIT
                 .ENDM
```

```
. . .
.ENDM
```

It is perfectly legal for a recursive macro to call another recursive macro and so on to any level you like. You need not to keep IF/ENDIF balance before exiting form the macro and the **xasm16** will do this automatically (the directive MACEXIT returns all conditionals to their original state).

6.9 Principle of Looking for Include Files

Firstly, Assembler looks for include files in source file path;

Secondly, Assembler looks for include files in current work path.

Example:

Source File Path: C:\SrcPath\SrcFile.asm

Current Work Path: C:\WorkPath

-I Path: .\INCLUDE

- Include incfile1.inc

  The first looking for path is "C:\SrcPath\incfile1.inc".

  The second looking for path is ".\INCLUDE\incfile1.inc" of which the absolute path is "C:\WorkPath\INCLUDE\incfile1.inc".

- Include OtherPath 1\OtherPath2\incfile1.inc The first looking for path is

  "C:\SrcPath\OtherPath1\OtherPath2\incfile1.inc".

  The second looking for path is ". \INCLUDE\OtherPath1\OtherPath2\incfile1.inc" of which the absolute path is "C:\WorkPath\INCLU DE\OtherPath 1\OtherPath2\incfile1.inc".

- Include \OtherPath 1\OtherPath2\incfile1.inc The first looking for

  path is "C:\OtherPath1\OtherPath2\incfile1.inc".

  The second looking for path is ".\INCLUDE\incfile1.inc" of which the absolute path is "C:\WorkPath\INCLU DE\OtherPath 1\OtherPath2\incfile1.inc".

- Include C:\OtherPath1\incfile1.inc

  The first looking for path is "C:\OtherPath1\incfile1.inc".

  The second looking for path is ".\INCLUDE\C:\OtherPath1\incfile1.inc" of which the absolute path is "C:\WorkPath\INCLU DE\C:\OtherPath 1\incfile1.inc".

7 Appendix C. Additional Linker Info.

7.1 Executable Output File Format

This output format is pure binary, with an assumed starting address of 0x0000. The linker fills gaps between the end of each section/file and the start of the next with "00" bytes when executable output file is .tsk format.

7.2 Linker Example

7.2.1 Prompt Mode

These examples illustrate how to use the linker in Prompt Mode. Responses to prompts are boldfaced.

All input, including null (default) responses, must end with a carriage return (<cr>).

- **Example 1a**

Suppose you have one assembly file exam.obj and you would like to execute output in TSK format, but there is no other option preference. In the command line, type Xlink16, and then the linker prompts and responses are:

```
Generalplus υnSP Linker – Ver. 1.5.0 (Build:1)
Input Body Name: SPCE500A<cr>
Input Filename: startup.obj<cr>
Input Filename: exam.obj<cr>
Input Filename: <cr>
Output Filename: exam.tsk<cr>
Library Filename: Cmacro.lib<cr>
Library Filename: <cr>
Options (D,C,M,X,3 <CR> = Default) : x<cr>
Enter Hex ROM Offset For 'CODE' in 'C:\CSTUDI~1\BIN\EXAM.OBJ' :
8000<cr>
Enter Hex RAM Offset For 'CODE' : <cr>
Enter Hex ROM Offset For 'DATA' in 'C:\CSTUDI~1\BIN\STARTUP.OBJ' :
8500<cr>
Enter Hex RAM Offset For 'DATA' : <cr>
Enter Hex ROM Offset For 'IRAM' : 8700<cr>
Enter Hex RAM Offset For 'IRAM' : 0<cr>
Enter Hex ROM Offset For 'RAM' : 0<cr>
Enter Hex RAM Offset For 'RAM' :<cr>
Enter Hex ROM Offset For 'sn_hwtest' :fc00<cr>
Enter Hex RAM Offset For 'sn_hwtest' : <cr>
Enter Hex ROM Offset For 'TEXT' : 9000<cr>
Enter Hex RAM Offset For 'TEXT' : <cr>
```

```
exam.tsk

0 Errors, 0 Warnings
```

The linker will read the file exam.obj and startup.obj, relocate the section with addresses that user input and output an executable file with the name exam.tsk.

- **Example 1b**

Suppose you have one assembly file exam.obj, and you would like to have an executable output file in Motorola s37 format and output High Level Symbol Table. The example uses the C option .In the command line, type Xlink16, and then the linker prompts and responses are:

```
Generalplus unSP Linker – Ver. 1.5.0 (Build:1)
Input Body Name: SPCE500A<cr>
Input Filename: startup.obj<cr>
Input Filename: exam.obj<cr>
Input Filename: <cr>
Output Filename: exam.tsk<cr>
Library Filename: Cmacro.lib<cr>
Library Filename: <cr>
Options (D,C,M,X,3 <CR> = Default) : 3,c<cr>
Enter Hex ROM Offset For 'CODE' in 'C:\CSTUDI~1\BIN\EXAM.OBJ': 8000<cr>
Enter Hex RAM Offset For 'CODE': <cr>
Enter Hex ROM Offset For 'DATA' in 'C:\CSTUDI~1\BIN\STARTUP.OBJ' :
8500<cr>
Enter Hex RAM Offset For 'DATA' : <cr>
Enter Hex ROM Offset For 'IRAM' : 8700<cr>
Enter Hex RAM Offset For 'IRAM' : 0<cr>
Enter Hex ROM Offset For 'RAM' : 0<cr>
Enter Hex RAM Offset For 'RAM' : <cr>
Enter Hex ROM Offset For 'sn_hwtest' : fc00<cr>
Enter Hex RAM Offset For 'sn_hwtest' : <cr>
Enter Hex ROM Offset For 'TEXT' : 9000<cr>
Enter Hex RAM Offset For 'TEXT' : <cr>
exam.s37

0 Errors, 0 Warnings
```

The linker will output an executable file with the name exam.s37 and High Level Symbol Table exam.sym.

- **Example 1c**

The same as the example 1b, except that you create a disk load map file here. In the command line, type Xlink16, then the linker prompts and responses are:

```
Generalplus unSP Linker – Ver. 1.5.0 (Build:1)
Input Body Name: SPCE500A<cr>
Input Filename: startup.obj<cr>
Input Filename: exam.obj<cr>
Input Filename: <cr>
Output Filename: exam.tsk<cr>
Library Filename: Cmacro.lib<cr>
Library Filename: <cr>
Options (D,C,M,X,3 <CR> = Default) : 3,c,m<cr>
Enter Hex ROM Offset For 'CODE' in 'C:\CSTUDI~1\BIN\EXAM.OBJ':8 000<cr>
Enter Hex RAM Offset For 'CODE' : <cr>
Enter Hex ROM Offset For 'DATA' in 'C:\CSTUDI~1\BIN\STARTUP.OBJ':
8500<cr>
Enter Hex RAM Offset For 'DATA' : <cr>
Enter Hex ROM Offset For 'IRAM' : 8700<cr>
Enter Hex RAM Offset For 'IRAM' : 0<cr>
Enter Hex ROM Offset For 'RAM' : 0<cr>
Enter Hex RAM Offset For 'RAM' : <cr>
Enter Hex ROM Offset For 'sn_hwtest' : fc00<cr>
Enter Hex RAM Offset For 'sn_hwtest' : <cr>
Enter Hex ROM Offset For 'TEXT' : 9000<cr>
Enter Hex RAM Offset For 'TEXT' : <cr>
exam.s37
0 Errors, 0 Warnings
```

The linker will output an executable file with the name exam.s37, a High Level Symbol Table exam.sym and a disk load map file exam.map.

- **Example 1d**

Suppose there are two assembly files exam.obj and test.obj. You want an executable output file in Motorola s37 format. In the command line, type Xlink16, and then the linker prompts and responses are:

```
Generalplus unSP Linker – Ver. 1.5.0 (Build:1)
Input Body Name: SPCE500A<cr>
Input Filename: startup.obj<cr>
Input Filename: exam.obj<cr>
Input Filename: test.obj<cr>
Input Filename: <cr>
Output Filename: exam.s37<cr>
Library Filename: Cmacro.lib<cr>
Library Filename: <cr>
Options (D,C,M,X,3 <CR> = Default) : 3<cr>
Enter Hex ROM Offset For 'CODE' in 'C:\CSTUDI~1\BIN\EXAM.OBJ' : 8000<cr>
Enter Hex RAM Offset For 'CODE' : <cr>
Enter Hex ROM Offset For 'CODE' in 'C:\CSTUDI~1\BIN\TEST.OBJ' : 8200<cr>
Enter Hex RAM Offset For 'CODE' : <cr>
Enter Hex ROM Offset For 'DATA' in 'C:\CSTUDI~1\BIN\STARTUP.OBJ' :
8500<cr>
Enter Hex RAM Offset For 'DATA' : <cr>
Enter Hex ROM Offset For 'IRAM' : 8700<cr>
Enter Hex RAM Offset For 'IRAM' : 0<cr>
Enter Hex ROM Offset For 'RAM' : 20<cr>
Enter Hex RAM Offset For 'RAM' : <cr>
Enter Hex ROM Offset For 'sn_hwtest' : fc00<cr>
Enter Hex RAM Offset For 'sn_hwtest' : <cr>
Enter Hex ROM Offset For 'TEXT' : 9000<cr>
Enter Hex RAM Offset For 'TEXT' : <cr>
exam.s37

0 Errors, 0 Warnings
```

7.2.2 Enhanced Data File Mode Example

To illustrate the differences between the two data file modes, the examples are the same as those for Prompt mode.

- **Example 2a**

The same as the example 1a. You should create a link file *exam.lik* with following contents:

Version: 1.10

Options: tsk

obj: "startup.obj"

obj: "exam.obj"

lib: "Cmacro.lib"

Output: "exam.tsk"

Locate: CODE in "EXAM.OBJ" at 8000

Locate: DATA in "STARTUP.OBJ" at 8500

Locate: IRAM at 8700 linkat 0

Locate: RAM at 0

Locate: sn_hwtest at fc00

Locate: TEXT at 9000

After creating link file, type command **"xlink16** *exam***.lik** –body SPCE500A –bfile body.dat" in the command line. The linker will read the file *exam***.obj** and *startup***.obj**, relocate the section with addresses that you specify in link file and output an executable file with the name *exam.***tsk**.

- **Example 2b**

The same linked for Motorola s37 output, you should create a link file exam.lik with following contents:

Version: 1.10

Options: m37, adhighlevel

obj: "startup.obj"

obj: "exam.obj"

lib: "Cmacro.lib"

Output: "exam.s37"

Locate: CODE in "EXAM.OBJ" at 8000

Locate: DATA in "STARTUP.OBJ" at 8500

Locate: IRAM at 8700 linkat 0

Locate: RAM at 0

Locate: sn_hwtest at fc00

Locate: TEXT at 9000

After creating link file, type command **"xlink16** *exam***.lik** –body SPCE500A –bfile body.dat" in the command line. The linker will read the file *exam***.obj** and *startup***.obj**, relocate the section with addresses that you specify in link file. The linker also outputs an executable file with the name *exam.***s37**and a High Level Symbol Table file *exam***.sym.**

- **Example 2c**

The same, except that you create a disk load map file here. You should create a link file *exam.lik* with following contents:

Version: 1.10

Options: m37, adhighlevel, map

obj: "startup.obj"

obj: "exam.obj"

lib: "Cmacro.lib"

Output: "exam.s37"

Locate: CODE in "EXAM.OBJ" at 8000

Locate: DATA in "STARTUP.OBJ" at 8500

Locate: IRAM at 8700 linkat 0

Locate: RAM at 0

Locate: sn_hwtest at fc00

Locate: TEXT at 9000

After creating link file, type command **"xlink16** *exam*.**lik** –body SPCE500A –bfile body.dat" in the command line. The linker will read the *fileexam.***obj** and *startup.***obj***,* relocate the section with addresses that you specify in link file. The linker also outputs an executable file with the name exam.s37, a High Level Symbol Table file *exam.***sym** and a disk map file *exam.***map***.*

- **Example 2d**

The same as the example 1d. You should create a link file *exam.lik* with following contents:

Version: 1.10

Options: m37

obj: "startup.obj"

obj: "exam.obj"

obj: "test.obj"

lib: "Cmacro.lib"

Output: "exam.s37"

Locate: CODE in "EXAM.OBJ" at 8000

Locate: DATA in "STARTUP.OBJ" at 8500

Locate: IRAM at 8700 linkat 0

Locate: RAM at 0

Locate: sn_hwtest at fc00

Locate: TEXT at 9000

After creating link file, type command **"xlink16** *exam***.lik** –body SPCE500A –bfile body.dat" in the command line. The linker will read the file exam**.obj** and *startup***.obj***,* relocate the section with addresses that you specify in link file and output an executable file with the name *exam***.s37.**

7.2.3 Automatic Mode Example

- **Example 3a**

The same as the example 1a. You should create an ARY file exam.ary with
following contents:

Options: tsk

obj: "startup.obj"

obj: "exam.obj"

lib: "Cmacro.lib"

Output: "exam.tsk"

Libpath: "c:\ide\lib"

After creating ARY file, type command "xlink16 –a *exam.***ary** –body SPCE500A –bfile *body.***dat***"* in the command line, the –a option represent automatic mode, the linker will read the *fileexam.***obj** and *startup.***obj***,* relocate the section automatically and output an executable file *exam.***tsk***.*

Note: Linker will locate sections those specified by .lik file to target address firstly.

- **Example 3b**

The same as the example 1b. You should create an ARY file exam.ary with following contents:

Options: m37, adhighlevel

obj: "startup.obj"

obj: "exam.obj"

lib: "Cmacro.lib"

Output: "exam.s37"

After creating ARY file, type command **"xlink16** –a *exam***.ary** –body SPCE500A –bfile body.dat" in the command line, the –a option represent automatic mode, the linker will read the file *exam***.obj** and *startup***.obj**, relocate the section automatically and generate a executable file *exam***.s37,** a High Level Symbol Table file *exam***.sym.**

- **Example 3c**

The same as the example 1c. You should create an ARY file exam.ary with following contents:

Options: m37, adhighlevel, map

obj: "startup.obj"

obj: "exam.obj"

lib: "Cmacro.lib"

After creating ARY file, type command **"xlink16** –a *exam***.ary** –body SPCE500A –bfile body.dat" in the command line, the –a option represent automatic mode, the linker will read the file *exam***.obj** and *startup***.obj**, relocate the section automatically. The linker will generate an executable file *exam***.s37,** a High Level Symbol Table file exam.sym and disk map file *exam***.map.**

- **Example 3d**

The same as the example 1d. You should create an ARY file exam.ary with following contents:

Options: m37

obj: "startup.obj"

obj: "exam.obj"

obj: "test.obj"

lib: "Cmacro.lib"

After creating ARY file, type command **"xlink16** –a *exam***.ary** –body SPCE500A –bfile body.dat" in the command line, the –a option represent automatic mode, the linker will read the file *exam***.obj,** *test***.obj** and *startup***.obj**, relocate the section automatically and generate an executable file *exam***.s37.**

8 Appendix D. ASCII

8.1 ASCII Table

Table 8-1

Character	Binary	Octal	Decimal	Hex
NUL	00000000	000	000	00
SOH	00000001	001	001	01
STX	00000010	002	002	02
ETX	00000011	003	003	03
EOT	00000100	004	004	04
ENQ	00000101	005	005	05
ACK	00000110	006	006	06
BEL	00000111	007	007	07
BS	00001000	010	008	08
HT	00001001	011	009	09
LF	00001010	012	010	0A
VT	00001011	013	011	0B
FF	00001100	014	012	0C
CR	00001101	015	013	0D
SO	00001110	016	014	0E
SI	00001111	017	015	0F
DLE	00010000	020	016	10
DC1	00010001	021	017	11
DC2	00010010	022	018	12
DC3	00010011	023	019	13
DC4	00010100	024	020	14
NAK	00010101	025	021	15
SYN	00010110	026	022	16
ETB	00010111	027	023	17
CAN	00011000	030	024	18

Character	Binary	Octal	Decimal	Hex
EM	00011001	031	025	19
SUB	00011010	032	026	1A
ESC	00011011	033	027	1B
FS	00011100	034	028	1C
GS	00011101	035	029	1D
RS	00011110	036	030	1E
US	00011111	037	031	1F
SP	00100000	040	032	20
!	00100001	041	033	21
“	00100010	042	034	22
#	00100011	043	035	23
$	00100100	044	036	24
%	00100101	045	037	25
&	00100110	046	038	26
’	00100111	047	039	27
(	00101000	050	040	28
)	00101001	051	041	29
*	00101010	052	042	2A
+	00101011	053	043	2B
,	00101100	054	044	2C
-	00101101	055	045	2D
.	00101110	056	046	2E
/	00101111	057	047	2F
0	00110000	060	048	30
1	00110001	061	049	31
2	00110010	062	050	32
3	00110011	063	051	33
4	00110100	064	052	34
5	00110101	065	053	35
6	00110110	066	054	36
7	00110111	067	055	37
8	00111000	070	056	38

Character	Binary	Octal	Decimal	Hex
9	00111001	071	057	39
:	00111010	072	058	3A
;	00111011	073	059	3B
<	00111100	074	060	3C
=	00111101	075	061	3D
>	00111110	076	062	3E
?	00111111	077	063	3F
@	01000000	100	064	40
A	01000001	101	065	41
B	01000010	102	066	42
C	01000011	103	067	43
D	01000100	104	068	44
E	01000101	105	069	45
F	01000110	106	070	46
G	01000111	107	071	47
H	01001000	110	072	48
I	01001001	111	073	49
J	01001010	112	074	4A
K	01001011	113	075	4B
L	01001100	114	076	4C
M	01001101	115	077	4D
N	01001110	116	078	4E
O	01001111	117	079	4F
P	01010000	120	080	50
Q	01010001	121	081	51
R	01010010	122	082	52
S	01010011	123	083	53
T	01010100	124	084	54
U	01010101	125	085	55
V	01010110	126	086	56
W	01010111	127	087	57
X	01011000	130	088	58

Character	Binary	Octal	Decimal	Hex
Y	01011001	131	089	59
Z	01011010	132	090	5A
[	01011011	133	091	5B
\	01011100	134	092	5C
]	01011101	135	093	5D
^	01011110	136	094	5E
_	01011111	137	095	5F
‘	01100000	140	096	60
a	01100001	141	097	61
b	01100010	142	098	62
c	01100011	143	099	63
d	01100100	144	100	64
e	01100101	145	101	65
f	01100110	146	102	66
g	01100111	147	103	67
h	01101000	150	104	68
i	01101001	151	105	69
j	01101010	152	106	6A
k	01101011	153	107	6B
l	01101100	154	108	6C
m	01101101	155	109	6D
n	01101110	156	110	6E
o	01101111	157	111	6F
p	01110000	160	112	70
q	01110001	161	113	71
r	01110010	162	114	72
s	01110011	163	115	73
t	01110100	164	116	74
u	01110101	165	117	75
v	01110110	166	118	76
w	01110111	167	119	77
x	01111000	170	120	78

Character	Binary	Octal	Decimal	Hex
y	01111001	171	121	79
z	01111010	172	122	7A
{	01111011	173	123	7B
|	01111100	174	124	7C
}	01111101	175	125	7D
~	01111110	176	126	7E
DEL	01111111	177	127	7F

8.2 ASCII Control Character Abbreviations

Table 8-2

Hex	Abbreviation	Character
00	NUL	Null or all zeros
01	SOH	Start of heading (^A)
02	STX	Start of text (^B)
03	ETX	Start of text (^C)
04	EOT	End of transmission (^D)
05	ENQ	Inquiry (^E)
06	ACK	Acknowledge (^F)
07	BEL	Bell (^G) (\a)
08	BS	Backspace (^H) (\b)
09	HT	Horizontal tabulation (^I) (\t)
0A	LF	Line feed (^J) (\n)
0B	VT	Vertical tabulation (^K) (\v)
0C	FF	Form feed (^L) (\f)
0D	CR	Carriage return (^M) (\r)
0E	SO	Shift out (^N)
0F	SI	Shift in (^O)
10	DLE	Data link escape (^P)
11	DC1	Device control 1 (^Q)
12	DC2	Device control 2 (^R)
13	DC3	Device control 3 (^S)

Hex	Abbreviation	Character
14	DC4	Device control 4 (^T)
15	NAK	Negative acknowledge (^U)
16	SYN	Synchronous idle (^V)
17	ETB	End of transmission block (^W)
18	CAN	Cancel (^X)
19	EM	End of medium (^Y)
1A	SUB	Substitute (^Z)
1B	ESC	Escape
1C	FS	File separator
1D	GS	Group separator
1 E	RS	Record separator
1F	US	Unit separator
20	SP	Space or blank (\040)
21	DEL	Delete

9 Appendix E. Struct And Procedure

9.1 Directive Introduction

STRUCT

Group: Definition

Function: Define the start of a

struct Syntax:

label:

.STRUCT Note:

Begins a struct definition.

Example: test1: .STRUCT

ENDS

Group: Definition

Function: Define the end of

a struct Syntax: *.ENDS*

Note: Terminates a stuct definition.

PROC

Group: Definition

Function: Define the start of a

procedure Syntax: label:

.PROC Note: Begins

a procedure definition.

Example: test1: .PROC

ENDP

Group: Definition

Function: Define the end of a

procedure Syntax: *.ENDP*

Note: Terminates a procedure definition.

9.2 Struct Definition

Syntax: Struct_name:**.STRUCT**

Storage

description **.E**

NDS

```
Example: test1:    .STRUCT
                   ad:     .DW 10;
                   bs:     .DW 'abcd';
                   gh:     .DD 0FFFCh;
                   .ENDS
```

; In this example, a struct type 'test1' is defined, ; it contains three child fields 'ad', 'bs', 'gh' and ; each has initial vale 10, 'abcd', 0FFFCh

9.3 Struct Variable Definition

Syntax: ***Struct_variable:*** ***.struct_name [expression_list]***

; The expression_list is used to store value to

; child fields of struct_variable.

```
Example: Stru_var1: .test1 [20, 'ad',7Dh]
         Stru_var2: .test1 [10,,7Dh]        // Not store a value to the second child field.
         ; So it keeps the initial value.
```

9.4 Struct Variable Reference

Syntax: ***Stru ct_va riable.chiled_ field***

```
Example:    R1 + = Stru_var1.ad    // 'Stru_var1' is struct variable defined above
                                   and
                                   // 'ad' is a child field defined above.
```

9.5 Procedure Definition

Syntax:

Proc
_name
:.PRO
C
Instructio
n_list
RETF;

.ENDP

```
Example:        qw:         .PROC
              label1:       R1 + = 20h;

                            CMP R1,80H

                            JL label1

                            RETF;

                            .ENDP
```

9.6 Procedure Reference

```
Syntax:         CALL              proc_name

Example:      CALL                sub1;       // sub1 is a procedure defined above
```

10 Appendix F. Limitation Of XASM

10.1 Labels

All label names, including macro name, section name, struct name, struct variable name and procedure name, are case-sensitive.

Non-local labels can have any numbers of characters, but only 32 characters are significant. A label can start in any column and its name must end with a colon.

All label names must start with an alphabetic character. They may not include any non-alphanumeric character except the underscore (_). The local labels are used as non-local labels. The assembler normally identifies a local label by a question mark (?) prefix or suffix.

10.2 Assembler Directive

Assembler directives are not case sensitive. You can type them in lower or upper case, or combination of both. Assembler directives cannot be used as labels.

10.3 Procedure Definition

Inside the definition of procedure, changing section is not allowed and directive END cannot be used.

10.4 Struct Definition

Inside the definition of struct, macro definition and macro reference are not allowed. A local label cannot be defined as a child field of a struct.

10.5 Logical, relation and bit operator

In XASM16, the logical, relation and bit operator symbols are same as ANSI C.

10.6 Reference of Symbols

In XASM16, forward reference is not allowed. If you reference a symbol that has not been defined already, it will generate an error message.

10.7 Code Size

In the object code, generated by a single .asm assembly, the user defined section in .CODE section or the inheritance of .CODE is restricted within 64K words locations.

11 Appendix G. Using Resource

1. After adding a resource file to IDE resource workspace, IDE will assign an ID for the file (If a data file DOG.16K, ID will be named for RES_DOG_16K).

   Note:

   To avoid resource ID conflicts in one project, do not save them with the same filename.

2. In compiling mode, resource compiler will transform the resource file into .obj format. Making label to both head and end of the file. The rule for label:

   If the file's ID is assigned RES_DOG_16K, the initiative label is '__RES_DOG_16K_sa', the end label is '__RES_DOG_16K_ea' and the section name is '__sn_section_RES_DOG_16K'

   Example: Using by asm code.

```
R1 = OFFSET __RES_DOG_16K_sa;
SR = SR AND 0x3f;                      // Keep original code segment
SR + = HIGH6 __RES_DOG_16K_sa;              // Adjust data segment
R2 = DS:[R1++];                             // Get 1st raw data
R3 = DS:[R1++];                             // Get 2nd raw data
…….
```

3. For conveniently, IDE will make a resource table for each resource. In compiling mode, add entry to "Resource.asm" and "Resource.inc". Making variable to both head and end of the resource. The rule for variable:

   If the file's ID is assigned RES_DOG_16K, the initiative variable is '_RES_DOG_16K_SA' and the end variable is '_RES_DOG_16K_EA'. (Note: Each entry will occupy four words)

   Example 1: Using by asm code

```
.include Resource.inc
R1 = _RES_DOG_16K_SA;
R2 = [R1++];
[DATA_START] = R2;   // Get data file offset
R2 = [R1];
R2 = R2 LSL 4;
R2 = R2 LSL 4;
R1 = SR AND 0x3f;    // Get original code segment
R1 = R1 + R2 LSL 2;
[DATA_SR] = R1;     // Get data file bank index
...
```

   Example 2: Using by C code

C Sample code

```
extern int SoundInit(unsigned long Speech_Start, unsigned long Speech_End);
extern int SoundDecode();
extern unsigned long RES_DOG_16K_SA,RES_DOG_16K_EA;
int retval;
void main()
{
   SoundInit(RES_DOG_16K_SA,RES_DOG_16K_EA);

   retval = 0;
   while (retval == 0)
   {
      retval = SoundDecode();
  }
  return;
}
```

Partial code of API SoundInit:

```
.Code
_SoundInit:     .PROC
                PUSH BP, BP TO [SP];
                BP = SP + 1;
                PUSH R2, R4 TO [SP];
                R1 = [BP + 3];
                R2 = [BP + 4];
                [DATA_START] = R1;
                R1 = SR AND 0x3F;
                R2 = R2 LSL 4;
                R2 = R2 LSL 4;
                R1 = R1 + R2 LSL 2;
                [DATA_SR] = R1;
                R1 = [BP + 5];
                R2 = [BP + 6];
                [DATA_END] = R1;
                [DATA_ENDDS] = R2;
                …
                .ENDP
```

12 Appendix H. unSP Coding Flow

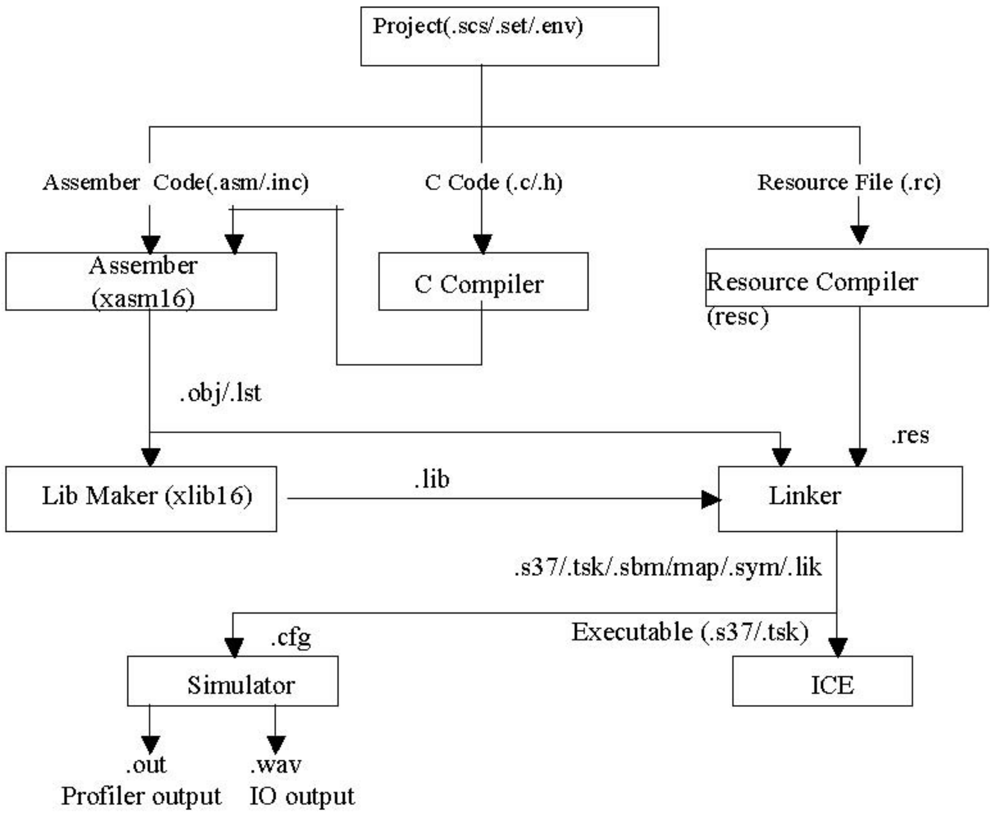

Figure 12-1