AST6/LST6
ASSEMBLER & LINKER
FOR THE ST6 FAMILY

User Guide

Release 1.0

May 1997

Table of Contents

Table of Contents

Table of Contents

Table of Contents

1 INTRODUCTION

AST6 is a macro-assembler that translates files that are written in assembler language into either executable files or object files. Executable files are files that are loaded into ST6 microcontrollers and can then be executed. Object files are intermediate files that you link together, forming a single executable file, using the LST6 linker. Whether you use AST6 to create an executable file, or create object files using AST6 then use LST6 to link them depends on your programming strategy, this is discussed later in this introduction.

AST6 and LST6 support the whole range of ST6 microcontrollers.

1.1 What is assembler language?

Assembler language is a symbolic code in which you develop applications. Symbolic code is made up of mnemonics and operands. Mnemonics are commands that have meaningful names, for example the ADD mnemonic adds two values together. Operands express complementary information to commands, such as addresses and values. You can also use meaningful names in operands. For example, a calendar application could use the symbolic name DATE for the current date. Using symbolic mnemonics and operands simplifies the application development process by letting you use meaningful names in your application. Files containing symbolic code are called source files.

Assembler programs are made up of the following elements:

- Machine instructions, or opcodes.
- Assembler Directives.

Machine instructions are codes that can be executed by the microcontroller without translation. Refer to the Databook for a full description of the machine instructions that are available for the ST6 microcontroller you are using.

Assembler directives control the assembly process. They can be used, for example, to define macros, or specify where in the microcontroller's memory, executable code and data are stored. The AST6 and LST6 directives are listed in "DIRECTIVES" on page 59.

The source files in which you develop your application, and thus enter directives and machine instructions, have the extension **.ASM**. You can write them using any ASCII text editor.

1.2 Programming Strategies

Before you start developing an ST6 application, you must decide:

- Whether you want to develop your program in either modular source files or a single source file.
- Whether or not you will use the paginated program space feature. This feature is described “Paged Program Memory” on page 26.

The choice you make determines the process you perform to generate the final, executable file.

1.2.1 Using Modular Source Files

Using modular source files means developing your program in a number of modules. Each module is held in one source file. The advantages of developing your program in modular source files are:

- Small programs are easier to debug, understand and maintain than large programs.
- You can test the output of a module in relation to the process it performs on the inputs.
- You can reuse modules in other programs.

1.2.2 Using Paged Program Space

The decision as to whether you use the paged program space is simple: if the final executable file will require more than 4 Kbytes of memory when loaded into the ST6, you must use the paged program space feature. Otherwise, you do not have to use this feature.

If you develop your program in modular source files, or if you use the paged program space feature, you must carry out the following steps in order to generate an executable file:

1 Assemble each of the source files that make up the program individually, using AST6. Assembled modular source files are called relocatable object files, and have the extension **.OBJ**. The word relocatable is used because the exact location that the generated object code will have in the ST6 memory is unknown. To generate these you must run AST6 with the **-O** option (see “Running AST6” on page 52).

2 Link the assembled object files into a single, executable file, using LST6. Executable files have the extension **.HEX**. The default file name generated by LST6 is ST6.HEX. You can change this using the **-O** option when you run LST6.

1.2.3 Using a Single Source File

If you are developing a small program, that does not exceed 4 Kbytes, the advantages of working with modular source files may not apply and you do not have to use the paged program space feature. In this case, it is simpler to develop your application in one module, since you can generate an executable file using AST6, without having to go through the linkage phase using LST6. The file generated by AST6 from a single-source file is called an absolute object, since you specify the exact location of the executable file in the ST6 memory using the .ORG directive (see “ORG - Set Program Origin” on page 74).

1.3 Debugging Executable files

Once you have generated your executable files, you can test and debug them using either the Windows-based ST6 program debugger, WGDB6, or the DOS-based ST6 program debugger, ST6NDB. Both debuggers simulate the behaviour of your program when it is loaded into an ST6 microcontroller using either ST6 Simulator or the ST6 HDS Emulator.

The ST6 Simulator is a program that simulates the execution of ST6 programs. It includes Wave Form Editor, that enables the simulation of ST6 pin input and output.

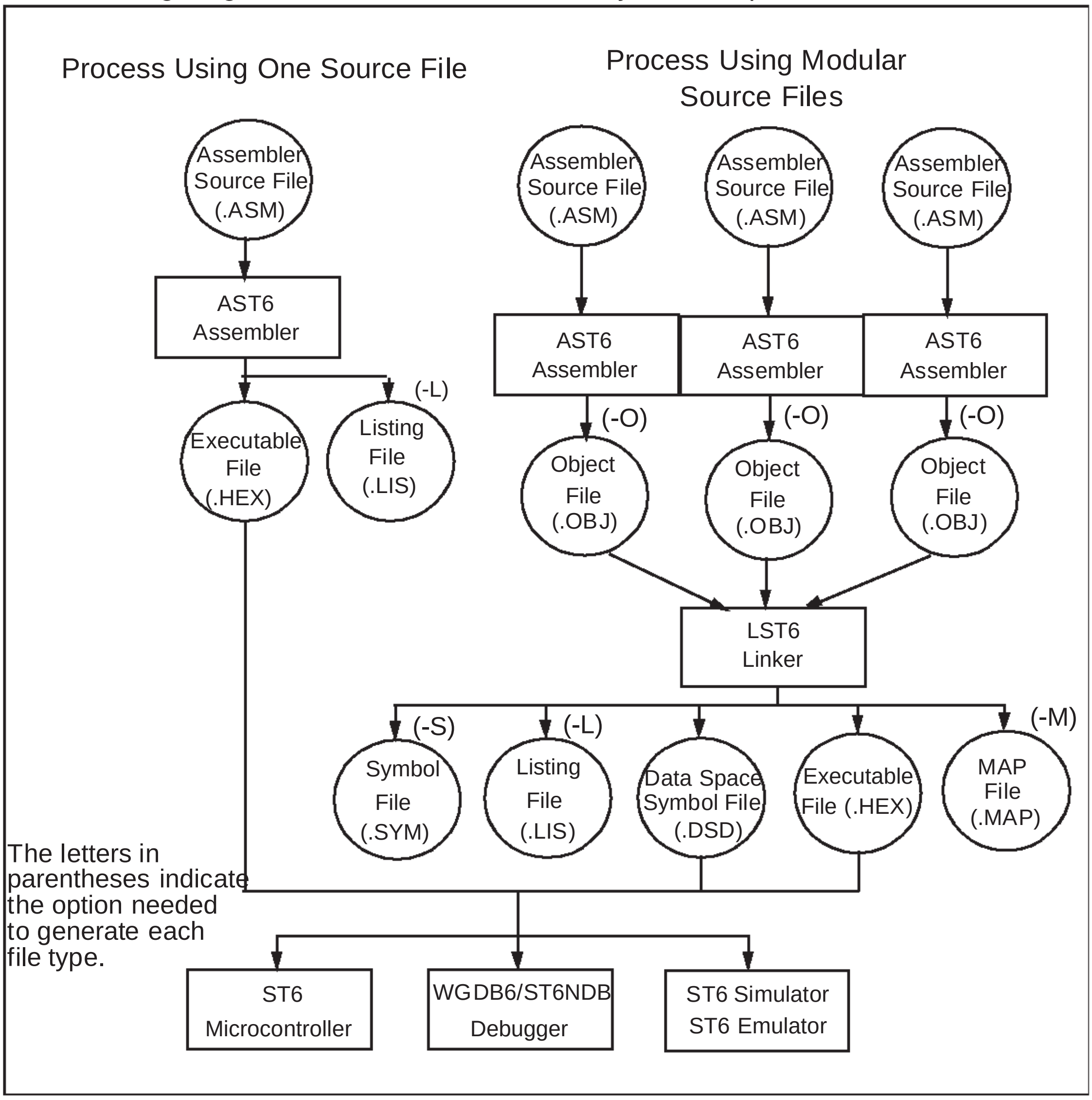
The ST6 HDS Emulator is a hardware system that enables real-time execution of ST6 applications.

Note that if you want to use either of the debuggers, you must generate **.DSD** and **.SYM** files during the assembly and link phases. “AST6 and LST6 Source and Generated Files” on page 16 describes all the files that are involved in the assembly and link processes.

1.4 Loading Executable Files into ST6 Microcontrollers

Once your program is ready, you can load it into ST6 microcontrollers using the EPROM programmer.

The following diagram summarises the assembly and link processes.



1.5 ST6 Memory Structure

The ST6 memory is divided into two principal components, the program space and the data space.

The program space is an area of ROM memory in which the instructions to be executed, the data required for immediate addressing mode instructions, and the user-defined vectors are stored. It is addressed using the 12-bit Program Counter register. ST6 microcontrollers that have more than 4 Kbyte ROM feature a paginated program space. In paginated ST6 program spaces, the area between real addresses 0 and 7FFh is paginated. The area between addresses 800h and FFFh on page 1 is static.

When referencing a page, the page is selected using the Program ROM Page Register (PRPR).

If you use program space pagination, you must structure your application accordingly, refer to “Working with the Program Space” on page 26 for further details.

Source code that is stored in the program space can be divided into sections. Sections are identified by a number, from 0 to 32. Each section starts at address 0 for the current module. Sections enable you to write source code in any order, but specify the order in which they are linked into the final assembled code. During the link edit phase, sections are allocated to pages. By default, LST6 allocates sections to pages by matching section numbers to page numbers, thus section 0 is allocated to page 0, section 1 is allocated to page 1, and so on. The default size of a section is 2 Kbytes, however you can modify this, as well as define which section is stored in each page, using the -P option when you run LST6. You can allocate any number of sections, from any source file, to a page in the program memory, provided their total size does not exceed that of the page, which is 2 Kbytes.

The data space is an area of RAM memory that stores all the data required by the program. It also stores the standard ST6 registers. ST6 microcontrollers that have more than 64 byte RAM feature a paginated data space. In paginated ST6 data spaces, the area between addresses 0 and 3Fh is paginated into 64-byte RAM and EEPROM pages. When referencing data in a paged area, the page is selected using the Data RAM/EEPROM Bank Register (DRBR).

To provide you with additional data space, ST62 and ST63 family chips let you store read-only data, such as look up tables and constants in the program space. The area of program space used for storing data space information is called a Data ROM Window.

1.6 Installing AST6/LST6

To install the AST6/LST6 software, put the diskette marked **ST6-SOFTWARE TOOLS** into your floppy drive and copy its contents to your hard disk.

2 GLOSSARY OF TERMS

absolute object file. An object file whose location in memory is defined in the source code using the .ORG directive. Absolute objects are can only be generated from programs that are coded in one source file.

addressing mode. In order to decrease the size of instructions, and thus the space they take in the program memory and the time needed to execute them, instructions have different addressing modes, based on the minimum addressing informationrequired for each instruction.

assembler language. A symbolic code in which you develop applications, and that is translated into object or executable files using an assembler.

AST6. The ST6 family macro-assembler that translates files that are written in assembler language into either executable files or object files.

conditional assembly. The use of conditions in source files, according to which the subsequent lines of code are or are not assembled. Conditional assembly can be used to generate different program versions or executable files for different ST6 microcontrollers from the same source file.

cross reference table file (.X). A file that lists the symbols used in a program, and specifies the numbers of the lines that define or reference each symbol.

Data RAM/EEPROM Bank registerr (DRBR). A register that selects the data page to be accessed by the subsequent instruction(s).

Data ROM Window. An area in the data space through which you can access read-only data, such as look up tables and constants, that is stored in blocks of up to 64 Kbytes in the program space.

Data ROM Window Register (DRWR). A register that, together with an instruction address, specifies the block of data in the program space to be accessed via the Data ROM Window.

data space. An area of RAM memory that stores all the data required by the program. and the standard ST6 registers.

data space symbol file (.dsd). Files that list the symbols in the data space. They are required by the ST6 debuggers.

directives. Commands that control the assembly process. They can be used, for example, to define macros, or specify where in the microcontroller's memory, executable code and data are stored.

dynamic pages. Virtual pages in the paged area of the program space. They are repetitions of the same area of ROM whose real address is 0 to 7FFh. Each dynamic page has a virtual address to distinguish it from the others.

emulator. A hardware device that simulates ST6 microcontrollers, enabling real-time execution of ST6 applications.

entry point. The address from which an executable file is written to ROM.

error report file (.err). A file to which error and warning messages that are generated during assembly and linkage are optionally written.

executable file (.hex). A file that is ready to be loaded to a microcontroller and executed. ST6 executable file are in the Inter-HEX format.

expression. A constant or symbol, or any combination of the two, separated by an arithmetic operator.

external label. Labels that are external to a module are those that are defined in another module.

global symbol. Symbols that are defined in one source module, but that can be used by others.

label. A meaningful name that you can use to specify a memory location or symbol.

linker memory map file (.map). Linker memory map files list the start, end and size of all the sections in the application and the start locations and sizes of the relocatable objects.

listing file (.lis). An ASCII text file that shows the lines of generated object code together with the source code they were generated from.

LST6. The ST6 family linker, that links relocatable objects (assembled source file modules) into a single, executable file that can be loaded into the ST6 memory.

machine instructions. Codes that can be executed by the microcontroller without translation. Machine instructions are also called opcodes.

macro. A sequence of assembler instructions and directives that can be inserted into the source program in place of the macro name. Macros enable you to simplify code and reduce code development time by reusing frequently-used functions.

macro-assembler. An assembler that includes macro generation capabilities.

map section. A section that can be included at the end of a listing file of an absolute object, that lists the name, type and size of each section.

mnemonic. An instruction, that is converted into machine code by the assembler. Mnemonics have meaningful names, for example the ADD mnemonic adds two values together.

object file (.obj). An intermediate file that you link together, forming a single executable file, using the LST6 linker (relocatable object).

opcode. See Machine Instructions.

operand. The part of an instruction line that specifies complementary information for the instruction, such as contents and symbols. Operands may contain:

- Numbers
- String and character constants
- Program Counter References
- Expressions

paging. A feature in which an area of both the data space and the program space is duplicated into 'pages'. Pages are not physical areas of memory, they are repetitions of the same area that are distinguished using virtual addresses. Paging is a way of increasing the size of the data and program spaces to beyond that of their addressable area.

Program counter. A 12-bit register that points to the address of the instruction currently being executed in the program space.

Program ROM Page Register (PRPR). A register that indicates the program space page to be accessed.

program space. The area of ROM memory in an ST6 microcontroller in which programs are stored.

relocatable object. The separately-assembled source files that make up a program. The word relocatable is used because the exact location that the generated object code will have in the ST6 memory is unknown.

ROM Masking. A process that involves manually filling all reserved and unused areas of ROM with a predefined value. ROM masking is recommended, since it improves the reliability of your program when it is executed in the microcontroller.

sections. Divisions of code enabling you to write source code in any order, but specify the order in which they are linked into the final assembled code. During the link edit phase, sections are allocated to pages. By default, LST6 allocates sections to pages by matching section numbers to page numbers, thus section 0 is allocated to page 0, section 1 is allocated to page 1, and so on.

source file (.asm). ASCII text file, in which you write program code. Source files are made up of lines, each of which is terminated by a new line characters. Each line may contain Labels, Mnemonics, Operands and Comments.

static area. The real addressable area of ROM. It includes two static pages:

- Page 1, which is the second page within the overlaid area.
- Page 32, which located in the area between addresses 0FF0h and 0FFFh, and is thus not in the paginated area.

symbol table file (.sym). A file that lists the value and type of each symbol in an assembled program. They are required by emulators.

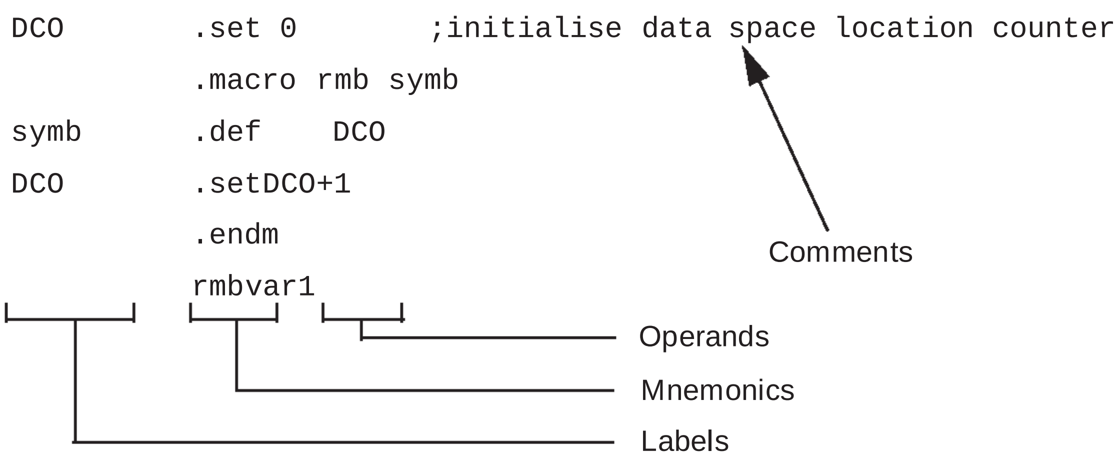
3 AST6 AND LST6 SOURCE AND GENERATED FILES

This section describes the format of source files that AST6 can assemble, and the output files that AST6 and LST6 generate either automatically or when an option is selected.

3.1 Source Files

AST6 source files have the extension **.asm**. .asm files are made up of lines, each of which is terminated by a new line characters.

Source files have the following format:



Each line may contain up to four types of information:

- Labels, which let you specify a memory location or symbol using a meaningful name.
- Mnemonics, which are instructions that are converted into machine code.
- Operands, which specify complementary information for an instruction, such as contents and symbols.
- Comments

These types of information must be entered in the above order. Each type of information must be separated by one or more spaces. The total width of a line can not exceed 400 characters. The following paragraphs describe these types of information.

3.1.1 Labels

Labels let you specify a memory location or symbol using a meaningful name. When a label is defined, it takes the current value of the address counter.

Labels must start in column one. A label may contain up to eight of any of the following characters:

- Upper case letters (A - Z)
- Lower case letters (a - z)
- Digits (0 - 9)
- Dollar sign ($)
- Underscore (_)

The first character of a label must be a letter or an underscore. Labels are case sensitive.

3.1.2 Mnemonics

Mnemonics must be separated from the preceding label (if there is one) by a space or a tab. Mnemonics specify the action to be performed by the assembler. Mnemonics can be the name of a machine instruction, an assembler directive code or a macro call. If a mnemonic is omitted from a line, the program counter is assigned to the label (if present).

3.1.3 Operands

Operands must be separated from mnemonics by one or more spaces. If more than one operand is used, the operands must be separated by commas. Operands may include:

- Numbers
- String and character constants
- Program Counter References
- Expressions

The following paragraphs describe these.

3.1.3.1 Numbers

The default radix for numbers is decimal. You can use numbers in other formats by following the number with the appropriate letter:

This letter:	Indicates this radix:
b or B	Binary
o or O	Octal
h or H	Hexadecimal

In hexadecimal, the decimal digits 10 - 15 are represented by upper or lowercase letters from A to F. Hexadecimal numbers that start with a letter must be preceded by the number 0. All numbers are defined as 16-bit signed values.

For example, the decimal value 45 is represented by 01000101b in binary, 55o in octal, and 2dh in hexadecimal.

3.1.3.2 String and Character Constants

String constants are strings of ASCII characters enclosed by double quotes. For example: “This is an ASCII string”. Character constants are single ASCII character enclosed by single quotes. For example ‘T’ .

3.1.3.3 Program Counter Reference

You can use the $ sign to identify the current value of the program counter (PC) in program space operands.

3.1.3.4 Expressions

Expressions in operands may contain numbers, labels or PC-relative references, separated by operators. Expressions are evaluated from left to right during assembly. Operators are evaluated according to their precedence, meaning that some operators are evaluated before others. Expressions within parentheses are evaluated first.

It is recommend that you use expressions containing program space symbols in jp/call instructions and variants of PC-relative instructions, such as jrr and jrs.

For example:

```
            ldi value, const1
            call subroutine1

subroutine1 ld A, value
            jrz out
            dec A
            jp subroutine1
out         ret
```

Such expressions are restricted to the following syntax:

expression = symbol

expression = symbol+constant_expression

expression = symbol-constant_expression

where constant_expression contains absolute references only.

The following table lists the available operators and their precedence.

Operator on operand	Meaning	Priority (the lowest value has highest priority)	Example
+	unary plus	1	+137
-	negation (2's complement)	1	-137
~	Bit inversion (1's complement)	1	~00111111 = 1100000
*	multiplication	2	38*3 = 114
/	division	2	114/3 = 38
%	modulo	2	38h%3 = 2
>> *n*	right shift*	2	
<< *n*	left shift*	2	
+	addition	3	038h+0FFh = 37h
-	subtraction	3	0FFh-038h = 0C7h
&	bitwise and	4	00001111&11111111 = 00001111
^	bitwise exclusive or	5	00001111^11111111 = 11110000
|	bitwise inclusive or	6	01001001|00010010 = 01011011

*Right shift and left shift shift the contents of the operand *n* places to the right or left respectively. For example:

```
sav_a       .def 08h
            ldi sav_a, 0FFh
            ldi A, sav_a >> 2       ; A=02h
            ldi A, sav_a << 2       ; A=20h
```

3.1.4 Comments

Comments are preceded by a semicolon. AST6 ignores all characters that follow a semicolon. Note that you can use semicolons in string and character constants.

3.2 Generated Files

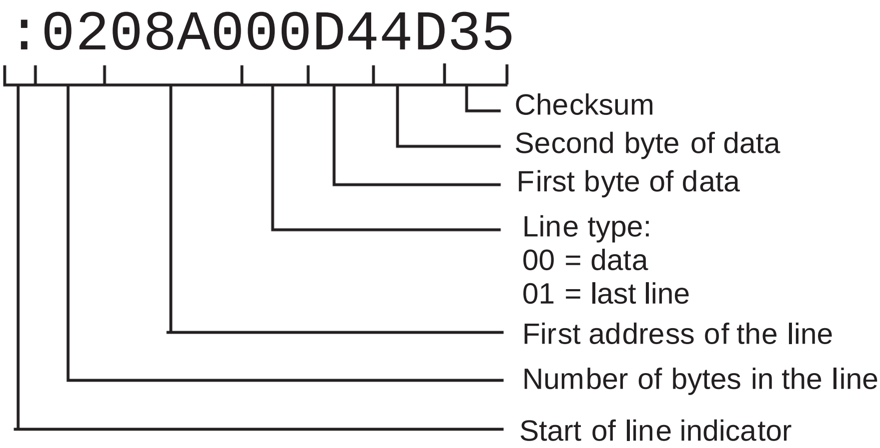
This section describes the files that are generated by AST6 and LST6.

3.2.1 Executable and Data Space Symbol Files

Executable (HEX) and data space symbol (DSD) files are automatically generated by AST6 if you run it without the -O option, or by LST6 if you use relocatable objects.

HEX files are in the INTEL-HEX format.

Below is an example line of a HEX file:



The checksum is calculated by starting at 0, then subtracting each byte from the previous result. Thus the total - the checksum = 0. For example:

00-02-08-A0-00-D4-4D-35=00

DSD files list the symbols in the data space. They are required by the ST6 debuggers.

3.2.2 Listing Files

Listing files show the lines of generated object code together with the source code they were generated from. To output a listing file, run AST6 with the **-L** option.

If you generate relocatable objects, you can update the listing files during the linking process by running LST6 with the **-l** option. Listing files are named <prog>**.lis**, where <prog> is the name of the assembled file.

Examples

To generate a listing file for absolute objects (single-source file programs):

`AST6 -L myprog` Generates the files myprog.lis, myprog.hex and myprog.dsd.

To generate a listing file for relocatable objects (modular file programs or programs that use program space paging):

`AST6 -L -O myprog1` Generates the files myprog1.lis and myprog1.obj.

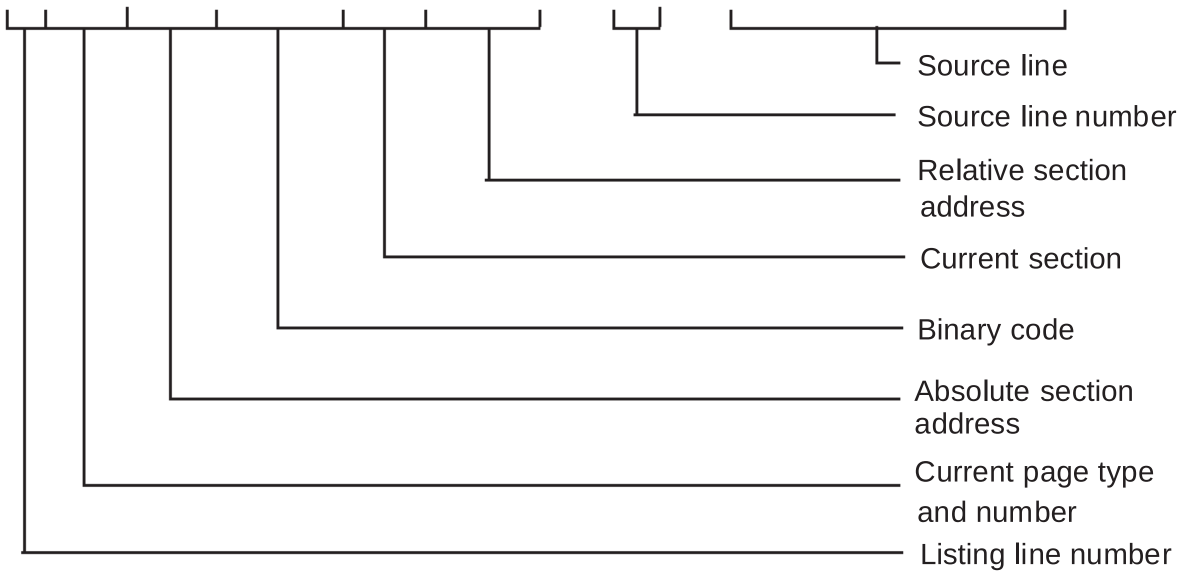
`AST6 -L -O myprog2` Generates the files myprog2.lis and myprog2.obj.

Then:

`LST6 -I -O myprog myprog1 myprog2` Updates the files myprog1.lis and myprog2.lis, and generates myprog.hex and myprog.dsd.

The following diagram shows an example AST6 listing file and describes what the various columns mean.

```
22 P00 0000 5F10   S00    0000      22     add    a,var1
23                                  23     .section1
24 P01 0800 BF20   S01    0000      24     and    a,var2
25 P01 0802 DF30   S01    0002      25     sub    a,var3
26                                  26
27                                  27     .section2
28 P02 1000 3F40    S02   0000      28     cp     a,var4
29 P02 1002 0D1000 S02    0002      29     clr    var1
30                                  30
31                                  31     mac_   ex1      var1
```



3.2.3 Including a Map Section

If you are using absolute objects (single-source file programs), you can include a map section at the end of listing files. If you are using relocatable objects, you can generate a separate map file using LST6 (see “Linker Memory Maps” on page 22).

The following diagram shows an example map section:

```
 ** SPACE 'PAGE_0' SECTION MAP **
______________________________________
|       name       |  type  |   size   |
|-----------------|--------|----------|
| PG0_0            |  TEXT  |      182 |
|_________________|________|__________|
Tue May 06 10:54:52 1997  file dummys.lis  page 19
** SPACE 'PAGE_1' SECTION MAP **
______________________________________
|       name       |  type  |   size   |
|-----------------|--------|----------|
| PG1_0            |  TEXT  |      158 |
|_________________|________|__________|
Tue May 06 10:54:52 1997  file dummys.lis  page 20
** SPACE 'PAGE_32' SECTION MAP **
______________________________________
|       name       |  type  |   size   |
|-----------------|--------|----------|
| PG32_0           |  TEXT  |       10 |
|_________________|________|__________|
```

The `type` column indicates the section type, this can be `text` for program space section or `data` for data space section.

To generate mapping information, run AST6 with the **-M** option as well as the **-L** option. For example:

`AST6 -L -M myprog` Generates the files myprog.lis, myprog.hex and myprog.dsd.

3.2.4 Linker Memory Maps

If you are using relocatable objects (modular file programs or programs that use program space paging), you can generate separate linker memory map files. Linker memory map files list the start, end and size of all the sections in the application and the start locations and sizes of the relocatable objects. Link process errors and warnings are also reported in linker memory maps.

Below is an example line of a linker memory map:

```
 *** ST6 Linkage Editor: 'dummys' object file Map ***
PROGRAM SECTIONS:
number  start   end     size
------  -----   ---     ----
0       0000    07FF    0182
1       0800    0F9F    014F
32      0FF0    0FFF    0010
WINDOW SECTIONS:
number  start   end     size
------  -----   ---     ----
0       0182    018A    0009
MODULE dummys.obj:
section    type    start    size
-------    ----    -----    ----
0          P       0000     0182
1          P       0800     014F
32         P       0FF0     0010
0          W       0182     0009
```

The `type` column in linker memory maps indicates the section type, this can be `P` for program space section or `W` for Data ROM Window section.

To generate a linker memory map, run LST6 with the **-M** option. The mapping file is named **ST6.MAP** by default. You can specify your own name by including the -**O** option when running LST6. In this case the file is named <prog>**.MAP**, where <prog> is the name of the assembled file. For example, the command:

`LST6 -M -O myprog myprog1 myprog2`

generates the files myprog.map, myprog.hex and myprog.dsd.

3.2.5 Cross Reference Tables

If you are using absolute objects (single-source file programs), you can generate cross-reference tables. These list, for each symbol, the numbers of the lines that define or reference that symbol. The line number that defines the symbol is followed by an asterisk (*). To generate a cross-reference table, run AST6 with the **-X** option.

Cross reference tables are named <prog>**.X**, where <prog> is the name of the assembled file. For example, the command:

```
AST6 -X myprog
```

Generates the file **myprog.X**

3.2.6 Symbol Table Files

Symbol table files list the value and type of each symbol in the assembled code. You must generate a symbol table file if you want to test your program using an emulator. Below is an example line of a symbol table file:

```
porta       :      EQU    00ff6h    P
```

Symbol Type
P = program space symbol
C = constant

Full 16-bit symbol address

Symbol name

If you are using absolute objects (single-source file programs), to generate a symbol table file, run AST6 with the **-S** option. AST6 symbol table files are named <prog>**.sym**, where <prog> is the name of the assembled file. For example, the command:

```
AST6 myprog -S
```

Generates the file **myprog.sym**

If you are usingrelocatable objects (modular file programs or programs that use program space paging), to generate a symbol table file, run LST6 with the **-S** option. The symbol table file is named **ST6.SYM** by default. You can specify your own name by including the -O option when running LST6. In this case the file is named <prog>**.SYM**, where <prog> is the name of the assembled file. For example, the command:

```
LST6 -S -O myprog myprog1 myprog2
```

generates the files myprog.map, myprog.hex and myprog.dsd.

If you run AST6 with the -O option (to generate a relocatable object), symbol table file generation is disabled, since in this case the program space symbols are defined in the link edit process.

3.2.7 Error Reports

By default, AST6 error and warning messages are displayed on your screen, and written to the listing file if you run AST6 with the **-L** option. You can choose to record error and warning messages in an error file, by running AST6 with the **-E** option. Error files are named <prog>**.err**, where <prog> is the name of the assembled file. For example, the command:

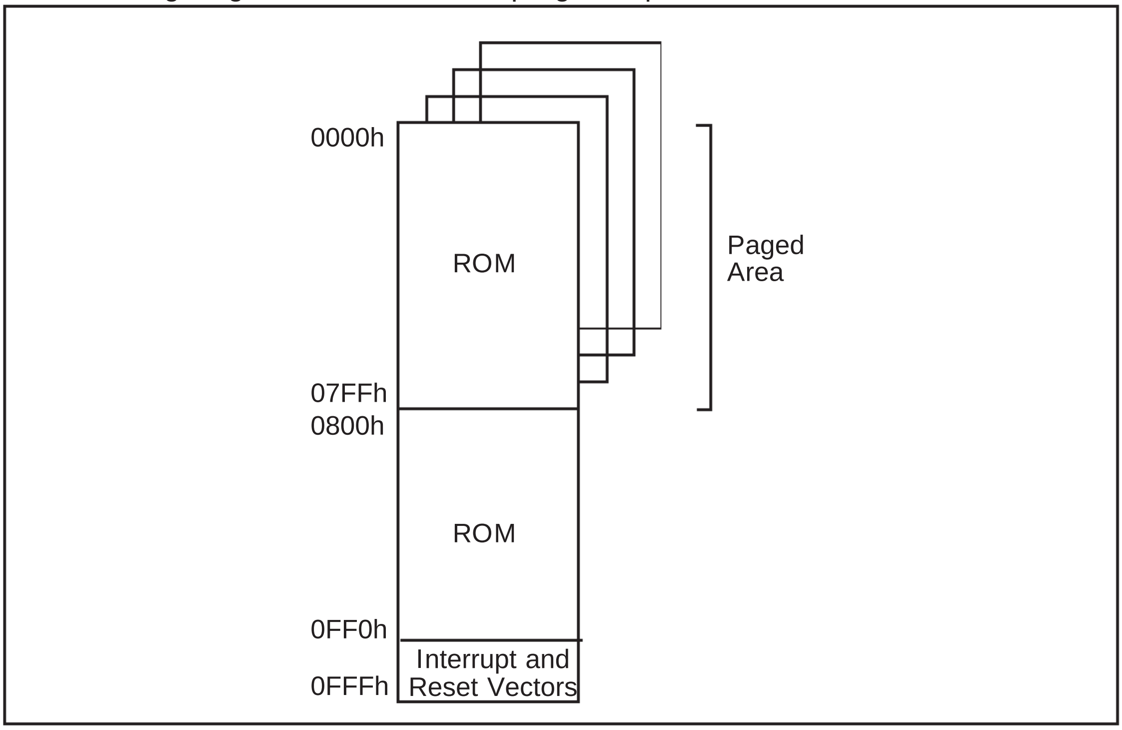
```
AST6 -E myprog
```

Generates the file **myprog.err**

LST6 writes errors to the file **stdout**.

4 WORKING WITH THE PROGRAM SPACE

The program space is an area of ROM memory in which the instructions to be executed, the data required for immediate addressing mode instructions, and the user-defined vectors are stored. It is addressed using the 12-bit Program Counter register. The following diagram shows the ST6 program space structure.



4.1 Paged Program Memory

ST6 microcontrollers that have more than 4 Kbytes of ROM feature a paginated program space.

This means that the ROM consists of a static area and up to 30 dynamic pages. Dynamic pages are virtual, they are repetitions of the same area of ROM whose real address is 0 to 7FFh. Each dynamic page has a virtual address to distinguish it from the others. Virtual address are allocated in relation to the page number, as shown in the table below.

The static area is the real addressable area of ROM. It includes two static pages:

- Page 1, which is the second page within the overlaid area.
- Page 32, which located in the area between addresses 0FF0h and 0FFFh, and is thus not in the overlaid area. Page 32 stores the interrupt and reset vectors.

It is better to think of pages 1 and 32 as areas of static ROM, although they are addressed as if they were pages.

To reference a page, the required page is selected using the Program ROM Page Register (PRPR).

You can perform jumps from the static area to any of the dynamic pages. You cannot, however jump directly from one dynamic page to another without first jumping to the static area. The following table shows the paged memory characteristics:

Page No.	Virtual Address	Real Address	Can jump to
0	0000 to 07FF	0000 to 07FF	Page 1
1	0800 to 0FEF	0800 to 0FEF	All pages
2	1000 to 17FF	0000 to 07FF	Page 1
3	1800 to 1FFF	0000 to 07FF	Page 1
n = 4 to 31	[n*800]-[(9n*80)+7FF]	0000 to 07FF	Page 1
32	0FF0 to 0FFF	0FF0 to 0FFF	All pages

The use of pages 0 and 2 to 31 are optional.

4.2 Developing Programs for the Paged Area

Source code that uses paged memory must be divided into sections. Each section is a block of code that can be allocated to a page during the link phase. Each section starts at address 0 for the current module. Developing programs in sections has the advantage that sections enable you to write source code in any order, but specify the order in which they are linked into the final assembled code. You can allocate any number of sections, from any source file, to a page in the program memory, provided their total size does not exceed that of the page, which is 2 Kbytes.

By default, LST6 allocates sections to pages by matching section numbers to page numbers, thus section 0 is allocated to page 0, section 1 is allocated to page 1, and so on. If you define more than once section with the same number, the sections are mapped to their appropriate pages contiguously, in the order in which their holding modules are listed when AST6 is executed.

The default size of a section is 2 Kbytes, however you can modify this, as well as define which section is stored in which page, using the **-P** option when you run LST6.

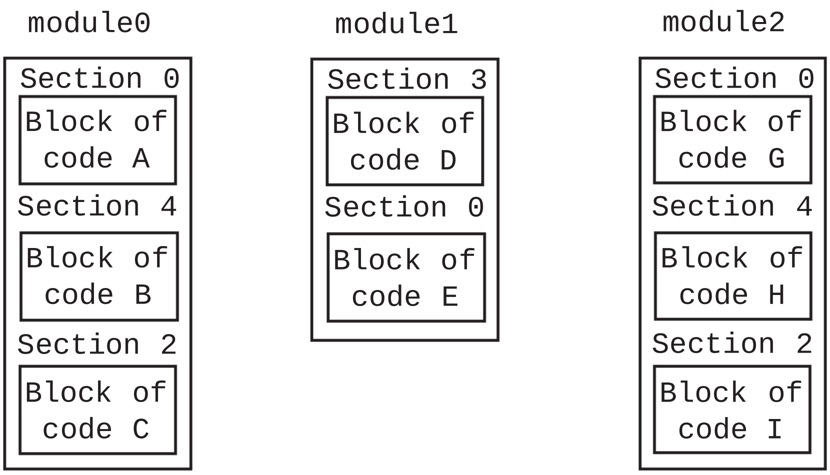
Allocating sections to pages using the **-P** option can be useful in two cases:

- For locating parts of the program, such as interrupt vectors, during the debugging phase.
- For limiting the memory space taken by final executable code and ensuring it is not written to any reserved areas of memory.

The **-P** option has the following format: **-P**<n>:<start>-<end>, where <n> is the section number, <start> is the start address and <end> is the end address. For example, to map section 1 to the area 400h to 7FFh you enter P1:400-7FF when you execute LST6.

You divide the module into sections using the .SECTION directive. The following diagram shows how LST6 allocates sections to pages when the -P option is not used:

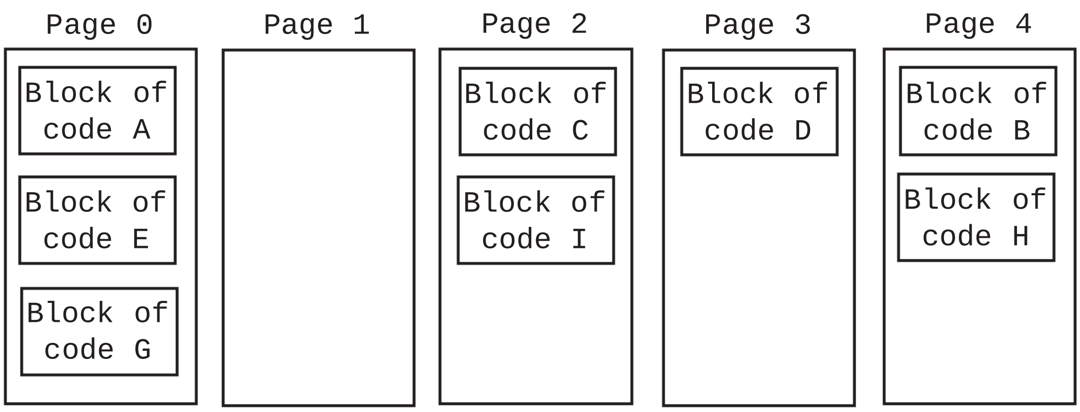
These modules:



Assembled as follows:

```
AST6 module0
AST6 module1
AST6 module2
```

Are mapped as follows:



You must assemble source files that use paged memory as relocatable objects, by executing AST6 with the -O option (see “Running AST6” on page 52).

Note: To be able to use this feature, you must include the PP_ON directive in your code before assembling it.

4.2.1 Accessing Paged Program Space

The Program ROM Page Register (PRPR) selects the page to be accessed. To simplify the use of the PRPR, you can use the <label>.P notation to load the location of the specified label to the PRPR. Thus, when jumping from one dynamic page to another, a jump is first made to page 1, where the <label>.P notation is used to load the target page. The jump is then made to the target. The following example shows how to program a jump from section 4 to section 5 (that are mapped to different pages during link editing):

```
             .pp_on
PRPR         .def 0cah          ; define PRPR
             .section 4
;            ...
             jp prs1            ;Jump to PRPR setter in page 1
caller       nop
             .section 1
             ...
prs1         ldi PRPR,target.p  ;set the page holding the label
                                ”target” in PRPR
             jp target          ;jump to the label ”target”
return       jp caller          ; return to calling section
;
             .section 5
;            ...
target       nop                ;Start the process
;            ...
             jp return          ;return to page 1
```

4.3 Using Absolute Objects

You can generate absolute objects if your program is made up of one module only and you are not using a paged program memory. To assemble an absolute object, you must execute AST6 without the **-O** option (see “Running AST6” on page 52).

When developing absolute object applications, you use the .ORG directive to specify the location of object code in the ST6 memory (see “ORG - Set Program Origin” on page 74). .ORG specifies the starting address or the subsequent code.

4.4 ROM Masking

ROM masking means manually filling all reserved and unused areas of ROM with a predefined value. ROM masking is recommended, since it improves the reliability of your program when it is executed in the microcontroller. To implement ROM masking, you must execute LST6, or AST6 if LST6 is not being used, with the -D option. By default, reserved and unused areas are filled with the value FFh. You can change this by specifying the value you want to use after the -D option (see the examples below). To enable AST6 or LST6 to perform ROM masking, you must provide the following information:

- The target ST6 type, by including the .VERS directive in your source file. See "VERS - Define Target ST6" on page 78.
- The size of the ROM in the target ST6, by including the .ROMSIZE directive in your source file. See "ROMSIZE - Set ROM Size for ROM Masking" on page 75.

Examples

The following command fills reserved and unused areas with the value 04h (the NOP instruction):

```
ast6 -d04 myprog
```

The following commands fill reserved and unused areas with the value FFh:

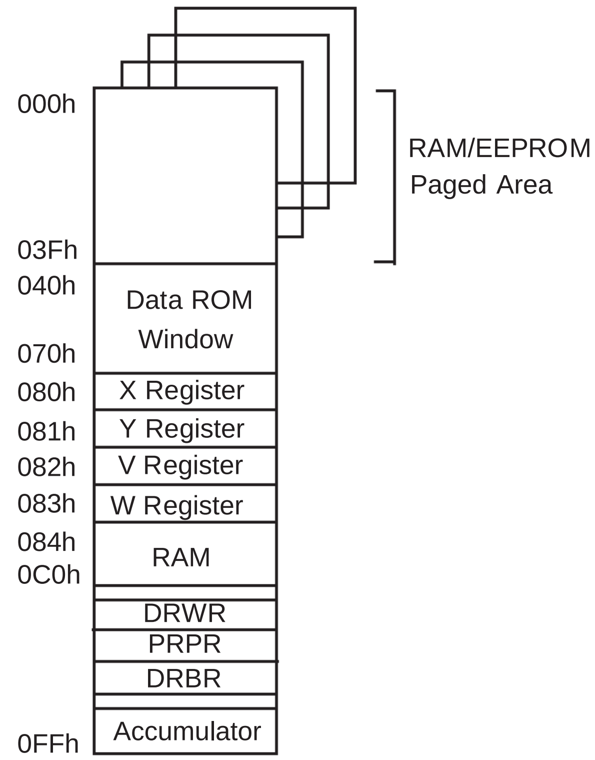
```
ast6 -O myprog
lst6 -d myprog
```

The following commands generate the file myprog.hex from myprog1.obj and myprog2.obj, and fill reserved and unused areas with the value 04h:

```
ast6 -O myprog1
ast6 -O myprog2
lst6 -d04 -O myprog myprog1 myprog2
```

5 WORKING WITH THE DATA SPACE

The data space is an area of RAM memory that stores all the data required by the program. It also stores the accumulator, indirect registers, short direct registers I/O port registers, the peripheral data and control registers, the Data ROM Window register and the Data ROM Window (see the Databook for the ST6 microprocessor you are using for further details of its memory configuration). The following diagram shows the structure of the data space:



You must define the characteristics of each byte that you want to use in the data space using the .DEF directive (see “DEF - Define Data Space Location Characteristics” on page 62). This includes the standard registers listed above. .DEF enables you to associate a label with an address and define the following characteristics:

- Read and write access.
- Its value.
- whether it is referenced in the .DSD file, which is referenced by the ST6 hardware emulator.

Therefore, all data space definition sections will always include the following lines defining the accumulator (A) and the Index registers (X, Y, V and W):

```
a             .def                0ffh, 0ffh, 0ffh
x             .def                80h, 0ffh, 0ffh
```

```
y            .def            81h, 0ffh, 0ffh
w            .def            82h, 0ffh, 0ffh
v            .def            83h, 0ffh, 0ffh
```

You cannot export data space symbol definitions, and thus share them with all the source modules that make up a program, using the .GLOBAL directive. You should therefore place all .DEF definitions in a separate file, that is included at the beginning of each source module using the .INPUT directive (see “INPUT - Read Source Statements from File” on page 68). An example of such a file is given in “Example Data Space Definitions File” on page 33.

Note that such multiple definition will cause a problem during the link edit phase: LST6 will find as many definitions of the same addresses as there are modules, and thus generate the appropriate error message. This problem can be overcome by preventing the multiple transmission of the definitions to LST6 using the .NOTRANSMIT and .TRANSMIT directives (see “TRANSMIT - Transmit Data Space Symbols to LST6” on page 77). You must, however allow the transmission of the definitions file for one module, so that its details are stored in the .DSD file.

The following example shows how to include a file named defs.h in the beginning of the source modules that make up an application:

```
;module 1
             .INPUT ”defs.h”
;...
;module 2
             .NOTRANSMIT
             .INPUT ”defs.h”
             .TRANSMIT
;...
;defs.h
             .pp_on
a            .def ffh
;...
```

An alternative approach is to create a macro for defining data space definitions. For example:

```
DCO          .set 0          ;initialise data space location
                             ;counter
```

```
            .macro rmb symb
symb        .def DCO
DCO         .set DCO+1
            .endm
            rmb var1
            rmb var2
```

5.1 Example Data Space Definitions File

The following example data definitions file defines the data space for an ST626x microcontroller.

```
;      *************************************
;      *  REGISTER/VARIABLE DECLARATION*
;      *************************************
x     .def 080h,0ffh,0ffh,m
y     .def 081h,0ffh,0ffh,m
v     .def 082h,0ffh,0ffh,m
w     .def 083h,0ffh,0ffh,m
a     .def 0ffh,0ffh,0ffh,m
IOR   .def 0c8h,0ffh,0ffh; Interrupt Option Register
DRWR  .def 0c9h,0ffh,0ffh; DATA ROM Window Register
;     **************
;     *   PORT A   *
;     **************
DRA   .def 0c0h,0ffh,0ffh; Data Register A
DDRA  .def 0c4h,0ffh,0ffh; Data Direction Register A
OPRA  .def 0cch,0ffh,0ffh; Option register A
;     **************
;     *   PORT B   *
;     **************
DRB   .def 0c1h,0ffh,0ffh; Data Register B
DDRB  .def 0c5h,0ffh,0ffh; Data Direction Register B
OPRB  .def 0cdh,0ffh,0ffh; Option register B
;     **************
;     *   PORT C   *
;     **************
```

```
DRC   .def 0c2h,0ffh,0ffh; Data Register C
DDRC  .def 0c6h,0ffh,0ffh; Data Direction Register C
OPRC  .def 0ceh,0ffh,0ffh; Option register C
;     **************
;     * A/D CONVER *
;     **************
ADCR  .def 0d1h,0ffh,0ffh; Control register
ADR   .def 0d0h,0ffh,0ffh; DATA register (result of conversion)
;     **************
;     *   TIMER    *
;     **************
;TSCR1.def 0d4h,0ffh,0ffh; TIMER STATUS control register
;TCR1 .def 0d3h,0ffh,0ffh; TIMER COUNTER register
;PSC1 .def 0d2h,0ffh,0ffh; TIMER PRESCALER register
;     *********************
;     * AUTO RELOAD TIMER *
;     *********************
ARMC  .def 0d5h,0ffh,0ffh; AR MODE control register
ARSC0 .def 0d6h,0ffh,0ffh; AR STATUS control register 0
ARSC1 .def 0d7h,0ffh,0ffh; AR STATUS control register 1
ARLR  .def 0d8h,0ffh,0ffh; AR LOAD register
ARRC  .def 0d9h,0ffh,0ffh; AR RELOAD/CAPTURE register
ARCP  .def 0dah,0ffh,0ffh; AR COMPARE register

WDR   .def    0d8h        ;watchdog register

psc   .def    0d2h,m
tcr   .def    0d3h,m
tscr  .def    0d4h,m

tmz   .equ    7
eti   .equ    6
tout  .equ    5
dout  .equ    4
psi   .equ    3
```

5.2 Paged Data Space

ST6 microcontrollers that have more than 64 bytes of RAM feature an optional paged data space. Thus, if your application requires more than 64 bytes of RAM, you can implement data space paging. In paged ST6 data spaces, the area between addresses 0 and 3Fh is paged into 64-byte RAM and EEPROM pages. When referencing a page, the required page is selected using the Data RAM/EEPROM Bank Register (DRBR).

To implement data space paging you must include the directive .DP_ON (see “DP_ON - Enable Data Space paging” on page 6312.2.7) in your source module.

5.2.1 Writing to Data Pages

The .PAGE_D directive defines the page to which subsequent data is written (see “PAGE_D - Specify Page Number for .DEF” on page 74). The data following a .PAGE_D directive is written to the page number specified by the directive. For example:

```
             .DP_ON
             PAGE_D 0
v1           .def 0
v2           .def 1
;...
             .PAGE_D 1
count        .def 0
colour       .def 1
;...
```

5.2.2 Accessing Data Pages

The page of data to be accessed is defined using the Data RAM/EEPROM Bank Register (DRBR). To avoid having to set DRBR each time you want to reference a data page, you can use the <label>.P notation, that sets DRBR to the data page holding the specified label.

The DRBR register selects the data page to be accessed according to the bit number (0 to 7) that holds a 1. The DRBR is implemented in different ways, depending on the ST6 you are using (see the Databook for the ST6 microprocessor you are using for further details).

The following example shows the use of <label>.p in selecting the data space page to be accessed.

```
                .DP_ON
RAMSW           .def 0e8h
a               .def 0ffh

                .PAGE_D 2
xx              .def 0
yy              .def 1
; ...
                .PAGE_D 1
                ldi RAMSW,xx.p      ;select data page
                                    containing xx
                ld a,xx
; ...
```

5.3 Using the Data ROM Window

To provide you with additional data space, ST62 and ST63 family microprocessors let you store read-only data, such as look up tables and constants, in areas of up to 64 bytes in the program space. Although it is physically located in the program space, the address of this area is 40h to 7Fh in the data space. This area is called the Data ROM Window.

To implement the Data ROM Window, you must include the .W_ON directive in the beginning of your source files. You can allocate any number of blocks of data to a continuous area of up to 64 bytes in the ROM. You can create as many 64-Kbyte blocks of data as you like within the ROM.

If you are generating relocatable object code, blocks of data to be stored in the Data ROM Window can be delimited using the .WINDOW and .WINDOWEND directives (see “WINDOW, WINDOWEND - Define Data Block in Program Space” on page 79). In this case, LST6 automatically defines the defined blocks of data as accessible via the Data ROM Window, in the order in which the modules are listed when LST6 is executed. It allocates blocks of data to spaces left free in the ROM after the program sections have been allocated. It does not necessarily use all the 64 bytes available for the Data ROM Window. An example of such an application is given in “Example Data ROM Window Application” on page 39.

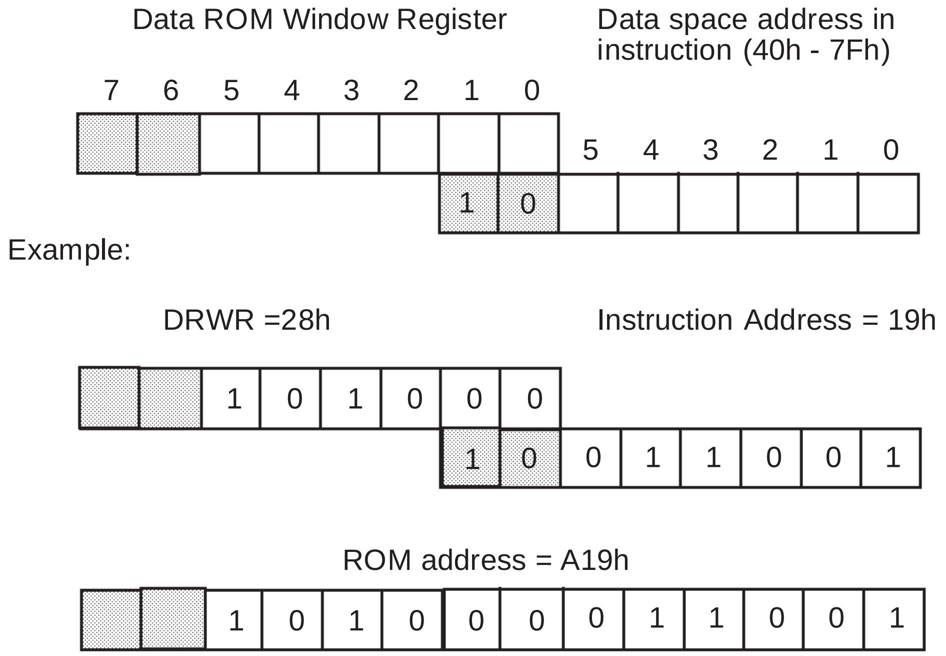
If you developing an absolute object you cannot delimit the window using .WINDOW and .WINDOWEND directives. In this case, you define the boundary of the block of data to be accessed using the Data ROM Window using the .BLOCK directive (see the example on page 39).

5.4 Accessing Data Within the Data ROM Window

The location of the block of data in the ROM to be accessed by the Data ROM Window is specified by the Data ROM Window Register (DRWR) and the address operand of the instruction accessing its contents.

Bits 5 to 0 of the DRWR define the start address of the block to be accessed via the Data ROM Window. Bits 5 to 0 of the address operand define the offset of the address to be accessed from the beginning of the block pointed to by DRWR. If the block of data to be accessed is within a ROM page, the PRPR must be used to specify the page holding the block, in the same that it is used to access any area of paginated ROM.

The following diagram shows how Data ROM Window addressing works.



5.4.1 Using <label>.D and <label>.W

To simplify the task of referencing data in the ROM via a Data ROM Window, AST6 includes two specific notations: <label>.D and <label>.W.

<label>.W enables you to set the DRWR to the block of data in ROM holding the specified label (see “LABEL.W - Initialise Data ROM Window Register” on page 70).

<label>.D enables you to set the offset to the specified label from the beginning of the block of data in ROM pointed to by the DRWR (see “LABEL.D - Access Data in Data ROM Window” on page 68). This is then used in the instruction address.

The following example shows how to access a constant, labelled CST1, that is held in a Data ROM Window:

```
LDI DRWR, CST1.W   ;Set the DRWR to the block of
                   ;data holding CST1
LDI X, CST1.D      ;Set the X register to the
                   ;address of CST1
LDI A, 40h         ;Load the value 40h into the
                   ;accumulator
ADDI A, X          ;Add the value held in CST1 to
                   ;the accumulator contents (40h)
```

Some more complete examples of how to use the Data ROM Window are given below.

Examples:

Using WINDOW and WINDOWEND to define a window and label to reference data within that window:

```
             .PP_ON             ;Must be executed for LST6
             .W_ON              ;Enables the use of windows
a            .def 0ffh
x            .def 80h
DRWR         .def 0cah          ;Define Data ROM Window register
             .WINDOW
cst2         .byte 22h
string2      .ascii ”ABCDEF”
;            ...
             .WINDOWEND
.section 2
             ldi DRWR,cst2.W    ;Select block holding cst2 and
                                ;string 2
             ld a,cst2.D        ;put the address of cst2 into a
             ldi x,string2.D    ;put address of string2 into a
```

In a single-module source :

```
                .PP_ON
                .W_ON               ;enables the use of windows
a               .def 0ffh
x               .def 80h
DRWR            .def 0cah           ;Define Data ROM Window register

                .section2
;               ...
                .block 64-$%64      ;Define 64-byte boundary
cst1            .byte 0ceh
string1         .ascii              ”abcdef”
;               ...
.section 0
ldi             DRW,cst1.W          ;select block holding cst1 and
                                    ;string 1
                ld a,cst2.D         ;put the address of cst2 into a
                ldi x,string2.D     ;put address of string2 into a
```

5.5 Example Data ROM Window Application

This example creates look-up tables in the ROM using the Data ROM Window. There are four 64-byte data tables, that are cascaded in order to provide a 256 byte non-linear correction table. For clarity, the table is applied to a linear 8-bit value, obtained from the ST6 on-chip analog-to-digital (a/d) converter. The example can easily be adapted for a wide range of applications, such as temperature sensing and control, frequency sensitivity correction, pattern generation and binary to bcd conversion.

To implement a 256-byte correction table, the two MSBs of the a/d result are used to reference one of the four 64-byte data tables. The remaining 6 LSBs of the result specify the offset from the beginning of the appropriate table.

```
;----- ST6 Table Look-up with Data ROM window
      .title  “tables.st6”
      .vers   “ST6215”
      .romsize 2
      .PP_ON                            ;enable linker
      .W_ON                             ;enable rom data window
;*********************
;standard definitions
;*********************
      .input  “c:\st6\input\std_def.st6” ;st6 standard def file
;*********************
;local definitions here
;*********************
tablemask       .equ    11000000b     ;mask for table number
offsetmask      .equ    00111111b     ;mask for offset value
rdw_start       .equ    040h          ;start of data-rom-window
watchtime       .equ    0ffh          ;watchdog timeout period
storeacc        .def    084h,0ffh,0ffh;store accumulator during INT
result          .def    085h,0ffh,0ffh,m;non-linear result storage
;**************
;initialisation
;**************
      .section 1
restart:
                reti                              ;ends reset condition
                                               ;enables nmi
      ldi     dwdr,#watchtime                  ;reload watchdog
      clr     a                                ;clear the accumulator
      set     ior4,ior                         ;enable interrupts
                                               ;configure port c
                ldi     drpc,#10h
                ldi     orpc,#10h
                ldi     ddrpc,#00h             ;pc4 is analog
;configure a/d
                set     pds,adcr               ;power up the a/d
```

```
      nop                                          ;allow a/d to settle
      ldi     adcr,#0b0h                           ;enable a/d interrupt
                                                   ;start conversion
;**************
;main code here
;**************
loop:           ldi     dwdr,#watchtime
                jp      loop                       ;continue
;***********
;subroutines
;***********
;*************************
;interrupt service routines
;*************************
ad_int:         ldi     dwdr,#watchtime
                ld      storeacc,a                 ;save accumulator
                ld      a,adr                      ;get a/d result
      ld      y,a                                  ;make another copy of a/d
                                                   ;result
      andi    a,#tablemask                         ;mask off lower six bits
                                                   ;acc. now contains table
                                                   ;number
testtab0:       cpi     a,#00000000b               ;table zero?
                jrnz    testtab1
                ldi     rdw,table0.w               ;point to table zero
                jp      offset
testtab1:       cpi     a,#01000000b               ;table one?
                jrnz    testtab2
                ldi     rdw,table1.w               ;point to table one
                jp offset
testtab2:       cpi     a,#10000000b               ;table two?
                jrnz    testtab3
                ldi     rdw,table2.w               ;point to table two
                jp offset
testtab3:       ldi     rdw,table3.w               ;point to table three
```

```
offset:                                     ;rdw now points to the
                                            ;correct table
      ld      a,y                           ;re-load a/d result
        andi    a,#offsetmask               ;mask off top bits
        addi    a,#rdw_start                ;add in rdw start address
      ld      x,a
;x now points to the correct value (in the correct table!)
      ld      a,(x)
      ld      result,a
;"result" now contains the non-linear value corresponding to the linear
;result obtained from the temperature measurement
      ldi     adcr,#0b0h                    ;start new conversion
      ld      a,storeacc                    ;recover accumulator
      reti
;*********************************
; timer interrupt service routine
;*********************************
tim_int:        reti
pbc_int:        reti
pa_int:         reti
nmi_int:        reti
;**************
; DATA TABLES *
;**************
      .window
table0:
      .byte   00h,00h,00h,00h,01h,01h,01h,01h
      .byte   02h,02h,02h,02h,03h,03h,03h,03h
      .byte   04h,04h,04h,04h,05h,05h,05h,05h
      .byte   06h,06h,06h,06h,07h,07h,07h,07h
      .byte   08h,08h,08h,08h,09h,09h,09h,09h
      .byte   0ah,0ah,0ah,0ah,0bh,0bh,0bh,0bh
      .byte   0ch,0ch,0ch,0ch,0dh,0dh,0dh,0dh
      .byte   0eh,0eh,0eh,0eh,0fh,0fh,0fh,0fh
      .windowend
```

```
        .window
table1:
        .byte   10h,10h,10h,11h,11h,11h,12h,12h
        .byte   12h,13h,13h,13h,14h,14h,14h,15h
        .byte   15h,15h,16h,16h,16h,17h,17h,17h
        .byte   18h,18h,18h,19h,19h,19h,1ah,1ah
        .byte   1ah,1bh,1bh,1bh,1ch,1ch,1ch,1dh
        .byte   1dh,1dh,1eh,1eh,1eh,1fh,1fh,1fh
        .byte   20h,20h,20h,21h,21h,21h,22h,22h
        .byte   22h,23h,23h,23h,24h,24h,24h,24h
        .windowend
        .window
table2:
        .byte   25h,25h,26h,26h,27h,27h,28h,28h
        .byte   29h,29h,2ah,2ah,2bh,2bh,2ch,2ch
        .byte   2dh,2dh,2eh,2eh,2fh,2fh,30h,30h
        .byte   31h,31h,32h,32h,33h,33h,34h,34h
        .byte   35h,35h,36h,36h,37h,37h,38h,38h
        .byte   39h,39h,3ah,3ah,3bh,3bh,3ch,3ch
        .byte   3dh,3dh,3eh,3eh,3fh,3fh,40h,41h
        .byte   42h,43h,44h,45h,46h,47h,48h,49h
        .windowend
        .window
table3:
        .byte   4ah,4bh,4ch,4dh,4eh,4fh,50h,52h
        .byte   54h,56h,58h,5ah,5ch,5eh,60h,62h
        .byte   64h,66h,68h,6ah,6ch,6eh,70h,72h
        .byte   75h,78h,7bh,7eh,81h,84h,87h,8ah
        .byte   8dh,90h,93h,96h,99h,9ch,9fh,0a2h
        .byte   0a6h,0aah,0aeh,0b2h,0b6h,0bah,0beh,0c2h
        .byte   0c6h,0cah,0ceh,0d2h,0d6h,0dah,0deh,0e2h
        .byte   0e6h,0eah,0eeh,0f2h,0f6h,0fah,0feh,0ffh
.windowend
;*******
;vectors
```

```
;*******
      .section 32                       ;section 0FF0h...
      jp      ad_int                    ;a/d interrupt vector
      jp      tim_int                   ;timer interrupt vector
      jp      pbc_int                   ;ports b&c interrupt vector
      jp      pa_int                    ;port a interrupt vector
      nop                               ;4 reserved bytes
      nop
      nop
      nop
      jp      nmi_int                   ;nmi interrupt vector
      jp      restart                   ;reset vector
      .end
```

6 IMPORTING AND EXPORTING LABELS

Global symbols are those that are defined in one source module, but that can be used by others. LST6 allows you to use two types of global symbol: labels that are defined in program sections and labels that are defined in Data ROM WIndows. You cannot use labels that are defined in the data space using the .DEF directive as global labels. Such labels should be defined in a separate file, that is included at the beginning of each source module using the .INPUT directive (see “Working with the Program Space” on page 26 for further datails on how to do this).

To specify a symbol that is defined by the current module, but will be referenced by other modules, use the .GLOBAL directive (see “GLOBAL - Define Symbols as Global” on page 67). A symbol must be defined as global before it is defined.

To specify a symbol that is referenced by the current module, but is defined by another module, use the .EXTERN directive (see “EXTERN - Define Symbols as External” on page 66). The following example shows how to import and export program section labels:

```
;module 1
             PP_ON

             .global label, cste
             .section 1
             ...
label:
             ...
             .block 64-$%64
cste:
;module 2
             .PP_ON
             .W_ON
a            .def 0ffh
DRWR         .def 0cah
             .extern label, cste
             .section 0
             ...
nop          jp label
```

```
                ...
.byte           ldi DRWR, cste.w
                ld a,cste.d
```

Note: Program Counter-relative jumps cannot be made to an external label.
LST6 checks that `label` is located in program page 1 (the static page), or in the same page as that in which it is referenced. If not, it will return an error message.

The following example shows how to import and export Data ROM Window labels:

```
;module 1
                .PP_ON
                .W_ON
                .global wc1, wc2
                .window
wc1             .byte 11h
wc2             .ascii “ABCDEF”
                ...
                .windowend
;module 2
                .PP_ON
                .W_ON
a               .def 0ffh
x               .def 80h
DRWR            .def 0cah
                .extern wc1, wc2
                .section 3
                ldi DRWR,wc1.W
                ld a,wc1.D
                ld x,wc2.d
                ...
```

Note: The <label>.D notation is used in module 2 because `wc1` and `wc2` are external and are thus assumed as being program section symbols.

7 DEVELOPING MACROS

Macros are sequences of assembler instructions and directives that can be inserted into the source program in place of the macro name. Macros enable you to simplify code and reduce code development time by reusing frequently-used functions.

You define the beginning and end of a macro using the MACRO and ENDM directives. For example, the following macro moves the contents of the cell pointed to by X one the next address, so that X points to the same data but at another address:

```
.MACRO Move1        ;Start of Move1 macro definition
ld A, (X)
inc X
ld (X), A
.ENDM               ;End of macro definition
```

Once you have defined a macro, you call it by including the macro name as you would any other mnemonic. For example, to call the above macro:

```
Move1
```

The macro is expanded in each place where its name is entered.

7.1 Nesting Macros

You can use two types of macro nesting: expansion nesting and definition nesting. Expansion nesting means calling, and thus expanding one macro from another macro. Definition nesting means defining and calling one macro from within the body of another macro.

An example of expansion nesting would be:

```
.MACRO Move2
Move1               ;Calls the macro Move1
ld A, (X)
inc X
ld (X), A
.ENDM               ;End of macro definition
```

In this case the body of macro Move1 is expanded within the body of macro Move2.

An example of definition nesting would be:

```
.MACRO Move2
.MACRO Move1        ;Start of Move1 macro definition
```

```
ld A, (X)
inc X
ld (X), A
.ENDM               ;End of Move1 definition
ld A, (X)
inc X
ld (X), A
.ENDM               ;End of macro definition
```

7.2 Macro Parameters

Macro parameters let you fill in values when you call a macro. They let you develop generic macros whose use can vary within the context of where it is expanded.

The parameters to be included with a macro are listed after the macro name, in the .MACRO directive. Multiple parameters must be separated by commas. For example, the following line creates the macro Move1 with the parameters Par1 and Par2:

```
.MACRO Move1 Par1,Par2
```

AST6 lets you use three types of macro parameter: normal parameters, numeric parameters and label parameters.

Normal parameters are substituted by a string of characters when the macro is expanded. For example, a normal parameter could hold a label to which the macro makes a jump.

Numeric parameters enable you to use symbols to specify numeric values. The parameter name must be a defined symbol. Numeric parameters are preceded by a backslash (\).

Label parameters automatically define label names when macro is expanded. If you specify a label directly within a macro, for example, for a loop within the macro body, if the same macro is called successively, the second call will generate a double-defined label error. Using a label parameter overcomes this problem. Label parameters are preceded by a question mark (?).

The following example demonstrates the use of these three types of parameter:

```
.macro zero start, \number, ?label
ldi x, start
ldi v, number
clr a
```

```
label        ld (x), a
             inc x
             dec v
             jrnz label
             .endm
```

This macro sets the number of bytes specified by `\number` to 0, from the start address specified by `start`. `?label` is replaced by the label specified when the macro is called. For example, the line:

```
             zero flagx, 5, here
```

Sets the 5 bytes starting at address `flagx` to 0, and uses the label `here`.

You can omit the label name when you call a macro. In this case AST6 generates its own names each time it expands the macro. The names generated are L01$, L02$, L03$ and so on.

7.3 Concatenating Symbols During Macro Expansion

AST6 enables you to concatenate two symbols during macro expansion. To concatenate two symbols, place the ’ operator between the two symbols you want to concatenate. You would concatenate two symbols, for example, to assign different symbols to a label when calling the same macro twice:

The following example demonstrates the use of the concatenation operator:

```
             .macro zero start, \number, ?lab
             ldi x, start
             ldi v, number
             clr a
sta’lab      ld (x), a
             inc x
             dec v
             jrnz sta’lab
             .endm
```

The line:

```
             zero flagx, 5, here
```

results in the label `stahere` being generated.

8 USING CONDITIONAL ASSEMBLY

AST6 lets you specify conditions, according to which the subsequent lines of code are or are not assembled. Conditional assembly can be used to generate different program versions or executable files for different ST6 microcontrollers from the same source file. Three directives enable you to program conditional assembly: IFC, ELSE and ENDC. They have the following format:

```
.IFC <condition> <argument1>...[<argumentn>]
...            ;Code to assemble if condition is true
.ELSE
...            ;Code to assemble if condition is true
.ENDC
```

where:

`<condition>` is one of the following conditions:

Condition	Meaning
EQ	If the following symbol = 0
NE	If the following symbol != 0
GT	If the following symbol >0
LT	If the following symbol <0
LE	If the following symbol <=0
GE	If the following symbol >=0
DF	If the following symbol is defined.
NDF	If the following symbol is not defined

`<argument1>...[<argumentn>]` are symbols or expressions to be subjected to the condition.

`.ENDC` identifies the end of the conditional assembler block.

Example

```
HTYPE        .SET 0
             .IFC EQ HTYPE
             NOP                ;assemble if HTYPE == 0
             .ELSE
             JP $
             .ENDC
```

9 APPLICATION DEVELOPMENT CHECKLIST

The following chart summarises the directives and options you must use in relation to the structure of your application, and the tasks you must carry out during the application development process.

In all programs...

Define data space characteristics using the .DEF directive.

Use the .DP_ON and .PAGE_D directives if using a paged data space.

Use the .W_ON and .WINDOW, .ENDWINDOW directives and LABEL.W, LABEL.D notations if using Data ROM Windows.

Use .VERS "ST62/63xx" and .ROMSIZE 2/4/8/16 and run AST6 with the -D option for ROM masking

in programs with one module...

and without paging...

Use the .ORG directive to locate the object file.

Use .ORG 0FF0h to map the exception vectors.

and with paging...

Include PP_ON at the beginning of the source file.

Use the .SECTION directive to locate the object file.

Map the exception vectors using .SECTION 32.

in programs with more than one module...

Include PP_ON at the beginning of each source file.

Use .SECTION and .WINDOW directives.

Use the -P option when running LST6 to allocate sections to precise addresses.

Map the exception vectors using .SECTION 32.

Use the -O option when running AST6.

Use LST6.

10 RUNNING AST6

To run AST6, enter the following command on the operating system command line.

```
AST6 [-<option1>...-<optionn>] <file1>[ <file2>... <filen>]
```

Where:

`option` is any of the following options:

Option:	Meaning:

C	Only write generated code to the listing file if the conditional directives are true. For example, the code:
	`ldi sav_a, 0FFh`
	`.ifc eq sav_a`
	`    ldi sav_a, 55h`
	`.else`
	`    clr sav_a`
	`.endc`
	Generates the following lines in the listing if the -C option is used:
	`ldi sav_a, 0FFh`
	`clr sav_a`
	If you omit this option, both generated and ignored lines of code are written to the listing file.

L	Creates a listing file named: `<file>.lis` (see “Listing Files” on page 20).
X	Create a cross reference table in `<file>.x` (see “Cross Reference Tables” on page 23).
M	Appends mapping information at the end of the listing files
S	Create a printable symbol table file in `<file>.sym` (see “Symbol Table Files” on page 24).
O	Use this option if your application includes more than one source file. Creates an object file in `<file>.obj`. Note that in this case the source files must be linked using LST6.
E	Create an error file in `<file>.err` (see “Error Reports” on page 24).

D[<pattern>]	Create the ROM mask (see “ROM Masking” on page 30). By default, unused and reserved ROM areas are filled with FFh. To use another value, enter a value in <pattern>. This option can be used without the directive .PP_ON. If you use option O, this option is turned off.
F	Include the full path name to the source file in error messages displayed on screen or stored in the .err file. For example, the message: ” warning example.asm 53: (91) r/w access not ...” with -F option becomes: ”warning c:\st62\kit624x\example.asm 53: (91) r/w access ”
W<level>	Changes the warning level that is traced. Enter the level you want to trace in <level> according to the table in “Warning Levels” on page 53.

<file1>[<file2>... <filen>] is the name of the source (.asm) files to be assembled.

10.1 Example

The command:

```
AST6 -L -S prog
```

Assembles **prog.asm**, generating **prog.hex** and creating a listing in **prog.lis**, a symbol table file in **prog.sym** and creating a file **prog.dsd** for the debugger.

To generate the correct files for the WGDB6 debugger, you must use either of the following options:

```
ast6 -L -O
```

and

```
lst6 -I -M -S -O
```

or

```
ast6 -L -S -M
```

10.2 Warning Levels

The following table lists the levels of warning that can be returned after AST6 has been executed:

This level:	Means this:
0	No errors encountered.
1	Warning(s) were encountered. These are either printed on screen or written to a .ERR file if the -E option was chosen. These are listed below.
2	Error(s) were encountered. These are either printed on screen or written to a .ERR file if the -E option was chosen. These are listed below.
3	There was an error on the command line.
4	System error(s) were encountered. These are related to the computer you are using, and not the assembly process.

10.3 AST6 Errors and Warnings

The following table lists the warnings that can be returned by AST6:

Level	Description
0	Minimum length assumed.
1	Symbol already imported.
1	Symbol already exported.
1	Symbol declared external but unused.
1	Both symbols have the same definition.
2	R/W access control not done on data-space operand.

11 RUNNING LST6

Note: When you run LST6, it creates up to 8 temporary files. On DOS systems, you may have to increase the default number of open files (refer to the configuration command descriptions in your MS-DOS documentation for further information).

To run LST6, enter the following command from the operating system command line.

```
LST6 [-<option1>...-<optionn>] <file1>[ <file2>... <filen>]
```

Where:

`option` is any of the following options:

Option	Meaning
F<Pattern>	Create the ROM mask (see “ROM Masking” on page 30). By default, unused and reserved ROM areas are filled with FFh. To use another value, enter a value in <pattern>. Note that this only works on programs that use the .WINDOW and .WINDOWEND directives. For other programs, use the -D option.
P<n>:<start>-<end>	Map the contents of program section <n> to the virtual addresses in the range <start>--<end>. See “Developing Programs for the Paged Area” on page 27 for further details.
E<name>	Assigns the entry point of the executable file to the global symbol specified in <name>. If this option is omitted, the entry point of the executable file is assigned to the start address of the program section <O>. The entry point value is specified in the last record of the .HEX output file.
J	Include each input module name before its local symbols in the symbol (.SYM) file. A pseudo-symbol is created: `<module name> EQU <order>` where `module name` is the file name, and `order` is the order in which the module was linked.
S	Creates a printable symbol table file in `<file>.sym` (see “Symbol Table Files” on page 24).
O <name>	Generate output files with the name specified in <name>. If this option is omitted the default name “ST6” is given.
M	Generate linker memory map. See “Linker Memory Maps” on page 22 for further details.
T[<list>]	Trace references to, and definitions of, the symbols listed in <list>. If <list> is omitted, all the global symbols are traced.

V	Displays link progress information messages, such as which object modules are loaded, and their sizes.
D<pattern>	Create the ROM mask (see “ROM Masking” on page 30). By default, unused and reserved ROM areas are filled with FFh. To use another value, enter a value in <pattern>. Note that if your program uses the .WINDOW and .WINDOWEND directives, you should use the -F option. If D is entered and the O option is omitted, the section numbers as defined by the .ROMSIZE and .VERS directives are used.
I	Updates the AST6 assembler listing files with the information that was modified during the link edit process.

`<file1>[ <file2>... <filen>]` is the name of the source (.obj) files to be assembled.

11.1 Using Parameter Files

Instead of re-entering the file names and options you want each time you run LST6, you can enter them in any type of ASCII text file, then call the file using the @ character. You can alos use text files to prevent the LST6 command exceeding the command line limit of 128 characters in DOS.

For example, to execute the command:

```
LST6 -S -0 myprog m1 m2 m3
```

You could enter:

```
LST6 @ params.txt
```

Where params.txt contains:

```
-S -0 myprog m1 m2 m3
```

11.2 Examples

The command:

```
LST6 -S -0 myprog m1 m2 m3
```

Links the modules **m1.obj**, **m2.obj** and **m3.obj**, generating the files: **myprog.HEX**, **myprog.DSD** and **myprog.SYM**.

The command:

```
LST6 -P0:000-3FF -P10:400-7FF
```

Places section 10 in program page 0, at offset 400h.

11.3 Errors and Warnings

All LST6 messages are output to the file **stderr** under Windows or **stdout** under DOS.

The following table lists the codes that are returned after LST6 has been executed:

This status:	Means this:
0	No errors encountered.
1	Warning(s) were encountered. These are listed below.
2	Error(s) were encountered. These are listed below.
3	There was an error on the command line. These are listed below.
4	System error(s) were encountered. These are related to the computer you are using, and not the assembly process.

11.4 Command Line Errors

The following table lists the error messages that can be returned after LST6 has been executed:

Error message	Meaning
bad option <x>	<X> is not a valid command line option.
bad argument <xx>	Incorrect argument <xx> following a valid option.
no input file	No input file was specified on the command line.
can’t open <file>	The file specified by <file> does not exist or read permission is denied.
conflicting start/end definitions <n> and <p>	The P option was used, and the sections <n> and <p> have overlapping start-end definitions. See “Developing Programs for the Paged Area” on page 27 for further details.
start/end definitions for section <n> not bounded on 2 k	A program section exceeds a 2-Kbyte page.
entry point <symbol> not in program space	An entry point was assigned to a symbol that does not exist in the program space.

11.5 LST6 Error Messages

The following table lists the error messages that are written to the file **stdout** under DOS.

Error message:	Meaning:
undefined symbol <symbol>	The symbol <symbol> is referenced as being external by a module, but is not defined. See “Importing and Exporting Labels” on page 45
multidefined symbol <symbol>	An imported or exported symbol name was repeated. See “Importing and Exporting Labels” on page 45.
section <n> overflow	Each program page is limited to 2048 bytes. While merging the contents of input files the maximum size was exceeded for section number <n>. See “Developing Programs for the Paged Area” on page 27 for further details.
not enough space in any used page to map window <n>	There was not enough space left in the program page for the specified window number.“Using the Data ROM Window” on page 36 for further details.
relocation overflow inside program section <n>, offset 0xHHH, <file>	The value of the external symbol, referenced at the specified offset, was too large to fit onto one byte or 3 hexadecimal digits.
type conflict relocating program section [window] <n>	An external reference was made to symbol definition that is not in the same type of section (program or window) in the referencing and referenced modules.
illegal jump inside section <n>	A jump was made to a label that was neither in the current dynamic page nor in the static page.
invalid type of relocation in program section [window]	The versions of LST6 and AST6 that were used are incompatible. Check the version numbers.
reserved symbol <symbol> already defined	You tried to redefine the listed symbol, which is reserved.
bad magic number <file>	The listed file is not compatible with LST6. I don’t know why it says magic number either.
memory allocation error <address>	Insufficient memory available to link a large module. You must divide it into smaller modules.
<file> bad object file format	Unexpected construct found in the listed file.
<file> premature end of file	The listed file cannot be read by LST6.
internal error (<comment>)	Either an invalid input file was used or an LST6 bug was encountered.

12 DIRECTIVES

This section describes the AST6/LST6 directives.

12.1 Directive Summary

The following table summarises the AST6 and LST6 directives.

Group	Action	Directive
Program Space Data Definition	Reserve a Block of Memory	[<label>] .BLOCK <expression>
	Generate Words of Object Code	[<label>] .WORD <expression>[,<expression>]
	Generate Bytes of Object Code	[<label>] .BYTE <expression>[,<expression>]
	Write Character String	[<label>] .ASCII ”<string>” [<label>] .ASCIZ ”<string>”
	Define ROM Size for ROM Masking.	.ROMSIZE <size>
	Begin ROM Code Section.	.SECTION <number>
Data Space Data Definition	Define Data Space Location Characteristics	[<label>] .DEF <address> ,[<R-mask>],[<Wmask>],[<value>][,<M|m>]
Data ROM Window directives	Enable Data ROM Windows	.W_ON
	Define Beginning of Data Block in Program Space	.WINDOW
	Define End of Data Block in Program Space	.WINDOWEND
	Initialise Data ROM Window Register	<label>.W
	Access Data ROM Window Data	<label>.D
Symbol Definition	Assign a Value to the Label	[<label>] .EQU <expression>
Linker Directives	Define Symbols as Global	.GLOBAL <symbol1>[,<symbol2>]...[,<symboln>]
	Transmit Data Space Symbols to the Linker	.TRANSMIT
	Don’t Transmit Data Space Symbols to the Linker	.NOTRANSMIT
	Define Symbols as External	.EXTERN <symbol1>[,<symbol2>]...[,<symboln>]

	Initialise Program ROM Page Register	<label>.P
Hardware-related directives	Enable Program Space paging	.PP_ON
	Enable Data Space paging	.DP_ON
	Specify Page Number for .DEF	.PAGE_D <number>
	Specify target ST6.	.VERS “<ST6>”
Miscellenaous directives	Display a String	.DISPLAY ”string”
	Define End of Source File	.END
	Read Source Statements from File	.INPUT ”filename”
	Set Program Origin	.ORG <expression>
	Generate Error Message	.ERROR ”string”
	Generate Warning Message	. WARNING ”string”
Listing directives	Insert Listing Page Eject	.EJECT
	Start/Stop Listing	.LIST 0 or 1
	Change Listing Lines Per Page	.PL <expression>
	Change Listing Characters Per Line	.LINESIZE <expression>
	Set Listing Page Header Title	.TITLE ”string”
	Insert Comment	.COMMENT <nn>
Conditional assembly directives	Begin Conditionally Assembled Code	.IFC <cond> <argument1>...[<argumentn>]
	Begin Alternative Assembled Code	.ELSE
	End Conditionally Assembled Code	.ENDC
Macro directives	Begin Macro Definition	[<label>] .MACRO macro_name [<par1>,...,<parN>]
	End a Macro Definition	[label] .ENDM
	End Macro Expansion	[label].MEXIT

12.2 Directive Descriptions

The following paragraphs describe the AST6 and LST6 directives.

12.2.1 ASCII, ASCIZ - Write Character String

Syntax

```
[<label>]    .ASCII ”<string>”
[<label>]    .ASCIZ ”<string>”
```

Description

Writes a character string to the program space. .ASCIZ is the same as .ASCII, except that it adds a NULL character to the end of the string.

Example

```
KDMESS       .ASCIZ ”1-Key Display”
```

12.2.2 BLOCK - Reserve a Block of Memory

Syntax

```
[<label>]    .BLOCK <expression>
```

Description

Reserves a block of memory in the program space. `<expression>` indicates the block dimensions. If `label` is included, the first address of the first memory location is assigned to it. All expressions used in the symbol must have been previously defined.

Example

To reserve a block of ROM memory to be accessed via the Data ROM Window, that starts at the beginning of a 64 Kbyte block and does not exceed 64 bytes:

```
             .block            64-$%64
```

12.2.3 BYTE - Generate Bytes of Object Code

Syntax

```
[<label>]    .BYTE <expression>[,<expression>]
```

Description

Generates successive bytes of object code in the program space, that contain the `<expression>` value in binary. The value of each expression is truncated to the first 8 bits.

Example

To generate a byte holding the value 0ceh (11001110).

```
.byte 0ceh
```

12.2.4 COMMENT - Set Comment Tabs

Syntax

```
.COMMENT <nn>
```

Description

Sets tabs in the comments that are printed in the listing. The tabs are set at the column number specified by the argument NN.

NN must be a positive decimal integer within either 256 or the value specified in .LINESIZE directive.

Example

To set a tab in column 50.

```
.comment 50
```

12.2.5 DEF - Define Data Space Location Characteristics

Syntax

```
[<label>] .DEF <address> ,[<R-mask>],[<W-mask>],[<value>][,<M|m>]
```

Description

Defines the characteristics of the specified location in the data space. This directive provides you with a useful tool for structuring the ST6 data space. If `<label>` is included, its value can be used in any place in the source file where a data symbol name can be used. You must define the characteristics of each byte that you want to use in the data space using this directive, even the standard registers. All the parameter values can be entered in any number base. Identifiers used in expressions and data addresses must be predefined.

`<R-mask>` specifies which bits can be read. If it is omitted, all the bits can be read. Each bit in `<R-mask>` that is set to 1 enables read access for the corresponding bit in the data space. For example, an `<R-mask>` value of 0FFh (11111111b) specifies that all the bits at the specified address can be read. 00Fh (00001111b) specifies that only bits 0-3 can be read.

<W-mask> specifies which bits can be written. If it is omitted, all the bits can be written. Each bit in <W-mask> that is set to 1 enables write access for the corresponding bit in the data space.

<M|m> places a marker in the .DSD file for this symbol, so that it can be viewed on screen during program simulation or emulation.

If a hardware register contains a mixed type of bits (Read/Write and Read-only/Write-only), R-mask and W-mask are defined according to the following convention:

- If a non-zero bit exists in the mask, the corresponding location is assumed to be accessible for read or write by a Load (LD) instruction.
- Rights are checked at bit level on bit test/set instructions.
- An immediate load (LDI) to a location will be checked against a “1” to a bit declared as non-accessible for write.

Example

To define a byte called val1 at address 000h that is write-only:

```
val1 .DEF     000h,0ffh,0
```

12.2.6 DISPLAY - Display a String

Syntax

```
.DISPLAY ”string”
```

Description

Displays the specified string on screen during the assembly process.

12.2.7 DP_ON - Enable Data Space paging

Syntax

```
.DP_ON
```

Description

Enables data space paging (in the data space address range 0-3Fh). See “Paged Data Space” on page 35.

This directive enables you to use the `.PAGE_D` directive and the notation `label.d` (with a label defined in data space).

12.2.8 EJECT - Insert Listing Page Eject

Syntax

```
.EJECT
```

Description

Inserts a new page eject into the listing file. The form feed character (^L) is sent to the printer and a new header is printed.

12.2.9 ELSE - Begin Alternative Assembled Code

Syntax

```
.ELSE
```

Description

Provides an alternative block of code to assemble if the IFC condition is not true. See also the IFC and ENDC directives.

See “Using Conditional Assembly” on page 50 for full details about conditional assembly.

Example

```
HTYPE       .SET 0
            .IFC EQ HTYPE
            NOP                ;assemble if HTYPE == 0
            .ELSE
            JP $
            .ENDC
```

12.2.10 END - Define End of Source File

Syntax

```
.END
```

Description

Defines the end of the source file. All lines after this directive are ignored by the assembler.

Note: This directive is not mandatory.

12.2.11 ENDC - End Conditionally Assembled Code

Syntax

```
            .ENDC
```

Description

Defines the end conditionally assembled code. See also the IFC and ELSE directives.

See “Using Conditional Assembly” on page 50 for full details about conditional assembly.

Example

```
HTYPE       .SET 0
            .IFC EQ HTYPE
            NOP               ;assemble if HTYPE == 0
            .ELSE
            JP $
            .ENDC
```

12.2.12 ENDM - End a Macro Definition

Syntax

```
[label]     .ENDM
```

Definition

Indicates the end of a macro definition. For full details about macros, see “Developing Macros” on page 47.

Example

```
            .MACRO Move
            ld A, (X)
            inc X
            ld (X), A
            .ENDM; Defines end of macro definition
```

12.2.13 EQU - Assign a Value to the Label

Syntax

```
[<label>] .EQU <expression>
```

Description

Assigns the value of the expression to the label. You cannot assign a value to the same label more than once using the .EQU directive. The symbols in the expression must be predefined. Note that you cannot define global symbols using this directive.

Example

To define the symbol Charge to a constant value (800):

```
Charge .EQU 800
```

12.2.14 ERROR - Generate Error Message

Syntax

```
.ERROR ”string”
```

Description

Generates the message `”string”` in the error file or the standard error output. See “Error Reports” on page 24 for further details.

12.2.15 EXTERN - Define Symbols as External

Syntax

```
.EXTERN <symbol1>[,<symbol2>]...[,<symboln>]
```

Description

To use this directive you must use the -O option on the command line when running AST6.

Defines the listed symbols as external. External symbols are not defined in the current module, but they are defined in another. See “Importing and Exporting Labels” on page 45 for further details.

Symbol names can not exceed 8 characters. This directive must be executed before the symbol is referenced.

Example

To use the symbol Charge, that was defined in another module, in the current module:

```
.EXTERN Charge
```

12.2.16 GLOBAL - Define Symbols as Global

Syntax

```
.GLOBAL <symbol1>[,<symbol2>],...,[,<symboln>]
```

Description

To use this directive you must use the -O option on the command line when running AST6.

Defines a symbol as global, thus it can be used by other modules. Symbol names must not exceed 8 characters. This directive must be executed before the symbol is defined. Symbols that were defined using the .SET, .DEF or .EQU directives cannot be defined as global. See “Importing and Exporting Labels” on page 45 for further details.

Example

To enable the symbol `Charge` to be used in another module:

```
.GLOBAL Charge
```

12.2.17 IFC - Begin Conditionally Assembled Code

Syntax

```
.IFC <condition> <argument1>...[<argumentn>]
```

where:

`<condition>` is one of the following conditions:

Condition	Meaning
EQ	If the following symbol = 0
NE	If the following symbol != 0
GT	If the following symbol >0
LT	If the following symbol <0
LE	If the following symbol <=0
GE	If the following symbol >=0
DF	If the following symbol is defined.
NDF	If the following symbol is not defined

`<argument1>...[<argumentn>]` are symbols or expressions to be subjected to the condition.

See “Using Conditional Assembly” on page 50 for full details about conditional assembly.

Example

```
HTYPE       .SET 0
            .IFC EQ HTYPE
            NOP                 ;assemble if HTYPE == 0
            .ELSE
            JP $
            .ENDC
```

12.2.18 INPUT - Read Source Statements from File

Syntax

```
            .INPUT ”filename”
```

Description

Reads the source statement(s) from the specified file. When the assembler reaches the end of the file, it returns to the calling source file. .INPUT directives can be nested. See “Working with The Data Space” on page 31 for a full example of how to use this command.

Example

To include a file named defs.h in the beginning of a source module:

```
;module 1
            .INPUT ”defs.h”
```

12.2.19 LABEL.D - Access Data in Data ROM Window

Syntax

```
<label>.D
```

Description

You can only use this notation after the `.W_ON` directive has been executed.

<label>.D lets you set the offset to the specified label from the beginning of the block of data in ROM pointed to by the DRWR.

Example

```
                .PP_ON
                .W_ON              ;Enable the use of windows
a               .def 0ffh
x               .def 80h
DRWR            .def 0cah          ;Define Data ROM Window register

                .section2
;               ...
                .block 64-$%64     ;Define 64-byte boundary
cst1            .byte 0ceh
string1         .ascii ”abcdef”
;               ...
                .section 0
                ldi DRWR,cst1.W    ;select window holding cst1 and
                                   string 1
                ld a,cst1.D        ;read cst1
                ldi x,string.D     ;point to address of string
```

Note: The arithmetic operations listed in “Expressions” on page 18 apply to <label>.D.

12.2.20 LABEL.P - Initialise Program ROM Page Register

Syntax

`<label>.P`

Description

The Program ROM Page Register (PRPR) selects the program space page to be accessed. The <label>.P notation enables you to load the location of the specified label to the PRPR. Thus, when jumping from one dynamic page to another, a jump is first made to page 1 (the static page), where the <label>.P notation is used to load the target page. The jump is then made to the target. For further details see “Accessing Paged Program Space” on page 29.

Note: If the specified label is in another module, the directive `.EXTERN LABEL` must be included before the use of `label.p`

Example

To jump from section 4 to section 5 (that are mapped to different pages during link editing) via page 1:

```
              .pp_on
PRPR          .def 0cah          ; define PRPR
              .section 4
;             ...
              jp prs1            ;Jump to PRPR setter in page 1
              .section 1
              ...
prs11         ldi PRPR,target.p  ;set the page holding
                                 the label ”target” in PRPR
              jp target          ;jump to the label ”target”
;
              .section 5
;             ...
target        nop                ;Start the process
```

12.2.21 LABEL.W - Initialise Data ROM Window Register

Syntax

`<label>.W`

Description

You can only use this notation in files that include the `.W_ON` directive.

The <label>.W notation sets the Data ROM Window Register (DRWR) to the block of data in the program space holding the specified label. <label>.W works on lables that are in the program space and in .WINDOW/.WINDOWEND blocks.

You can then reference data in the Data ROM window using its label, or <label>.D. See “Using the Data ROM Window” on page 36 for further details on the Data ROM Window.

Example

```
              .WINDOW
cst2          .byte 22h
```

```
string2         .ascii              "ABCDEF"
;               ...
                .WINDOWEND

                .section 2
                ldi DRWR,cst2.W     ; select block holding cst2 and
                                    string 2
                ld a,cst2           ; read cst2
                ldi x,string2.D     ; point to the address of
                                    string2
```

12.2.22 LINESIZE - Change Listing Characters Per Line

Syntax

```
.LINESIZE <expression>
```

Description

Changes the number of characters per line of the output listing to the value of `expression`. The default value is 131, the minimum value is 79.

Example

To set the output listing to 90 characters:

```
.LINESIZE 90
```

12.2.23 LIST - Start/Stop Listing

Syntax

```
.LIST 0 or 1
```

Description

.LIST 1 generates the listing file contents up to the point that .LIST 0 is reached.

Example

```
.LIST 1
...; these lines are written to the output listing
.LIST0
...; these lines are not written to the output listing
```

12.2.24 MACRO - Begin Macro Definition

Syntax

```
[<label>] .MACRO macro_name [<par1>,...,<parN>]
```

Definition

IMacros are sequences of assembler instructions and directives that can be inserted into the source program in place of the macro name.

`macro_name` is the name of the macro. Once a macro is defined, it is expanded in each place where its name is entered.

`par1 ... parN` are macro parameters, which let you fill in values when you call the macro. They let you develop generic macros whose use can vary within the context of where it is expanded.

The MACRO directive defines the beginning of a macro definition. For full details about macros, see “Developing Macros” on page 47.

Example

The following macro moves the contents of the cell pointed to by X one space further, so that X points to the same data but at another address:

```
.MACRO Move1       ;Start of Move1 macro definition
ld A, (X)
inc X
ld (X), A
.ENDM              ;End of macro definition
```

12.2.25 MEXIT - End Macro Expansion

Syntax

```
[label]      .MEXIT
```

Definition

lEnds macro expansion before its end is reached. This directive is normally used in a conditional assembly block (see “Using Conditional Assembly” on page 50).

Example

```
type         .set 0             ;Set type to 0
             .MACRO Move1 type  ; Defines start of macro Move1
             ...                ;(macro lines that are always
                                ;expanded)
             .IFC EQ type
             .MEXIT             ;End expansion if type == 0
             .ENDC
             ...                ;(macro lines that are expanded
                                ;if type <>0)
             .ENDM              ; End of macro definition
```

12.2.26 NOTRANSMIT - Don’t Transmit Data Space Symbols to LST6

Syntax

```
             .NOTRANSMIT
```

Description

To use this directive you must use the -O option when running AST6.

TRANSMIT transmits data space symbols to LST6 so that they are common to all modules. Notransmit turns off data space symbol transmission to LST6. These directives aim to prevent the same symbol from being defined twice in the .DSD file that is produced by LST6. This problem only occurs when if the program is split into several modules. The symbols defined in the data space by the .DEF directive must only be transmitted once to LST6.

Define all the common data space symbols in one module without using the TRANSMIT and NOTRANSMIT directives, and use these directives when you define data space symbols in other modules.

Example

Module 1.asm

```
        .input ”ST6STD.ASM”;Common data space symbol
                           ;definition file
```

Module 2.asm

```
        .notransmit
        .input ”ST6STD.ASM” ;
        .transmit
```

12.2.27 ORG - Set Program Origin

Syntax

```
        .ORG <expression>
```

Description

Sets the program origin for subsequent code to the address defined in `<expression>`. All symbols which appear in `<expression>` must have been previously defined. This directive can only be used when AST6 is called to produce an absolute object (without the **-O** option).

`.ORG` only applies to program space sections.

Example

To locate the subsequent code in the memory area starting at address 200h.

```
        .ORG 200h
```

12.2.28 PAGE_D - Specify Page Number for .DEF

Syntax

```
        .PAGE_D <number>
```

Description

Specifies the data memory page number to which subsequent data definitions using the .DEF directive apply (in the data space address range 0-3F). The page number must be in the range 0 - 255.

Example

To place v1 and v2 in data page 0:

```
        .DP_ON
        PAGE_D 0
v1      .def 0
v2      .def 1
```

12.2.29 PL - Change Listing Lines Per Page

Syntax

```
.PL <expression>
```

Description

Changes the number of lines per page on the output listing to the value of `expression`. The default value is 63 and the minimum value is 10. The first six and last six lines of a listing are empty.

Example

To set the number of lines in a listing page to 70:

```
.PL 70
```

12.2.30 PP_ON - Enable Program Space paging

Syntax

```
.PP_ON
```

Description

Enables program space paging. This directive enables use of the .SECTION directive and the notation <label>.P.

You must include this directive at the beginning of source files for ST62 and ST63 series devices whose program space exceeds 4k bytes. If you do not use this directive, you can only use the first 4 bytes in the program space.

12.2.31 ROMSIZE - Set ROM Size for ROM Masking

Syntax

```
.ROMSIZE n
```

Description

You must run AST6 with the -D option to be able to use this directive.

This directive sets the size of the ROM for ROM masking.

`n` defines the size of the microcontroller ROM, which must be one of the values 2, 4, 8 or 16.

Example

```
.ROMSIZE 2
```

12.2.32 SECTION - Begin Code Section

Syntax

```
.SECTION <number>
```

Description

Specifies the section number in which the subsequent code is placed. `<number>` specifies the section number, in the range 0-32.

You can only use this directive after the `.PP_ON` directive. Since the paged memory area (0 to 7FFh) is structured into overlayed pages, each page has a virtual address to distinguish it from the others. Virtual address are allocated in relation to the page number, as shown in the following table:

Page No.	Virtual Address	Real Address
0	0000 to 07FF	0000 to 07FF
1	0800 to 0FEF	0800 to 0FEF
2	1000 to 17FF	0000 to 07FF
3	1800 to 1FFF	0000 to 07FF
n = 4 to 31	[n*800]-[(9n*80)+7FF]	0000 to 07FF
32	0FF0 to 0FFF	0FF0 to 0FFF

During the link edit phase, sections are allocated to pages according to their numbers, thus section 0 is allocated to page 0, section 1 is allocated to page 1, and so on. You can allocate any number of sections, from any source module, to a page in the program memory, provided their total size does not exceed that of the page.

For further details about the use of sections and paged program space, see "Paged Program Memory" on page 26.

When LST6 is not used, the assembler implements three default sections: sections 0, 1 and 32.

Example

Module 1

```
              .PP_ON
              .SECTION 1
lab1          ldi  a,3
              .SECTION 2
              WAIT
              .SECTION 1
```

```
                NOP
```

Module 2

```
                .SECTION 1
                STOP
```

12.2.33 SET - Assign a Value to the Label

Syntax

```
[<label>]    .SET <expression>
```

Description

Assigns the value of the expression to the label. You can reassign values to label using the .SET command. The symbols in the expression must be predefined.

Example

To define the symbol Charge to the value 800:

```
Charge       .SET 800
```

12.2.34 TITLE - Set Listing Page Header Title

Syntax

```
             .TITLE ”string”
```

Description

Sets the title that is printed on output listing page headers.

12.2.35 TRANSMIT - Transmit Data Space Symbols to LST6

Syntax

```
             .TRANSMIT
```

Description

To use this directive you must use the -O option when running AST6.

TRANSMIT transmits data space symbols to LST6 so that they are common to all modules. Notransmit turns off data space symbol transmission to LST6. These directives aim to prevent the same symbol from being defined twice in the .DSD file that is produced by LST6. This problem only occurs if the program is split into several modules. The symbols defined in the data space by the .DEF directive must only be transmitted once to LST6.

Define all the common data space symbols in one module without using the TRANSMIT and NOTRANSMIT directives, and use these directives when you define data space symbols in other modules.

Example

Module 1.asm

```
.input ”ST6STD.ASM”; common data space symbol
                     definition
```

Module 2.asm

```
.notransmit
.input ”ST6STD.ASM” ;
.transmit
```

12.2.36 VERS - Define Target ST6

Syntax

```
.VERS ”<string>”
```

Description

Defines the ST6 type that the executable file will be loaded into. You must use this directive at the beginning of all source files. You specify the microcontroller name in `<string>`. If the microcontroller name includes a letter, omti the letter from the name, for example, for an ST62E25 enter ST6225.

Example

```
.VERS ”ST6210”
```

12.2.37 W_ON - Enable Data ROM Windows

Syntax

```
.W_ON
```

Description

You must use this directive at the beginning of the source file if you want to use the Data ROM window (see “Using the Data ROM Window” on page 36 for further details on the Data ROM Window).

12.2.38 WARNING - Generate Warning Message

Syntax

```
. WARNING ”string”
```

Description

Generates the message `”string”` in the error file or the standard error output. See “Error Reports” on page 24 for further details.

12.2.39 WINDOW, WINDOWEND - Define Data Block in Program Space

Syntax

```
.WINDOW
.WINDOWEND
```

Description

WINDOW defines the beginning of a block of data, stored in the program space, that can be accessed via the Data ROM window. WINDOWEND defines the end of the block of data.

You can only use these directives in source modules for relocatable objects (thus, use the -O option on the command line when running AST6).

If you are developing a source file to create an absolute object, you must define the boundary of the block of data using the `.block` directive.

These directives can only be used after the .W_ON directive (see “W_ON - Enable Data ROM Windows” on page 78). The directives you can use in with the Data ROM Window are: .BYTE, .WORD, .ASCII, .ASCIZ and .BLOCK.

For full details about Data ROM windows see “Using the Data ROM Window” on page 36. An example application that uses the .WINDOW and .WINDOWEND directives is provided in “Example Data Space Definitions File” on page 33.

Example

```
.PP_ON; must be executed if the linker is used
             .W_ON; enables the use of windows
a            .def 0ffh
x            .def 80h
DRW          .def 0cah; Define Data ROM Window register
```

```
                .WINDOW; Subsequent code is allocated to the window
cst2            .byte 22h
string2         .ascii          ”ABCDEF”
;               ...
                .WINDOWEND; End of code allocated to the window
```

12.2.40 WORD - Generate Words of Object Code

Syntax

```
[<label>]    .WORD <expression>[,<expression>]
```

Description

Generates successive 2-byte words of object code in the program space, that contain the `<expression>` value in binary. Words are stored in reverse order, thus the LSB has the lower address.

Example

```
val1         .WORD 0A0FFh
```

Index

Symbols

A

B

C

D

Index

P

R

S

T

V

W

NOTES: