A51
8051 Cross-Assembler

Version 0.49 or later

LIMITED WARRANTY

TRADEMARKS

Systronix®, Inc.
555 South 300 East #21
Salt Lake City, UT 84111
TEL: 801-534-1017
FAX: 801-534-1019
Internet: www.systronix.com
email: info@systronix.com
support: support@systronix.com

Revised - October 31, 1996

Revision History
1996 Oct 30 Improved description of DW directive

A51

Release 0.49 or later

Table of Contents

USING THE A51 CROSS-ASSEMBLER

This documentation assumes knowledge of assembly language programming and the 8051 instruction set. It is not a tutorial, and is not intended to teach 8051 assembly language programming. Its purpose is to enable you to use the A51 assembler by itself, to explain A51 syntax and formats, and to elaborate on A51 error, warning, and fatal error messages.

First, the question, "What does a cross-assembler do?" needs to be addressed as there is considerable confusion on this point. A cross-assembler is just like any other assembler except that it runs on some CPU other than the one for which it assembles code. For example, A51 assembles 8051 source code into 8051 object code, but it runs on an MS-DOS PC using the 8086 family CPU. The reason that cross-assemblers are useful is that you probably already have a CPU with memory, disk drives, a text editor, an operating system, and all sorts of hard-to-build or expensive facilities on hand. A cross-assembler allows you to use these facilities to develop code for another target CPU.

This program requires one input file (your 8051 source code) and zero to two output files (the listing and the object). The input file MUST be specified, or the assembler will terminate with a fatal error. The listing and object files are optional. If no listing file is specified, no listing is generated, and if no object file is specified, no object is generated. If the object file is specified, the object is written to this file in "Intel hexadecimal" format.

The command line for the cross-assembler looks like this:

```
A51 source_file [ -l list_file } [ -o object_file ]
```

where the [ ] indicates that the specified item is optional.

Example:

```
a51 test51.asm                        source:   test51.asm
                                      listing:  none
                                      object:   none

a51 test51.asm -l test51.prn          source:   test51.asm
                                      listing:  test51.prn
                                      object:   none

a51 test51.asm -o test51.hex          source:   test51.asm
                                      listing:  none
                                      object:   test51.hex

a51 test51.asm -l test51.prn -o test51.hex
                                      source:   test51.asm
                                      listing:  test51.prn
                                      object:   test51.hex
```

The order in which the source, listing, and object files are specified does not matter. Note that no default file name extensions are supplied by the assembler.

■INPUT FILE FORMAT

The source file that the cross-assembler processes into a listing and an object is an ASCII text file that you can prepare with whatever editor you have at hand. The most-significant (parity) bit of each character is cleared as the character is read from disk by the cross-assembler, so editors that set this bit should not bother this program. All printing characters, the ASCII TAB character (09H), and carriage-return linefeed combinations are processed by the assembler. All other characters are passed through to the listing file, but are otherwise ignored.

The source file is divided into lines by carriage-return linefeed characters. The internal buffers of the cross-assembler will accommodate lines of up to 255 characters.

Each source line is made up of three fields: the label field, the opcode field, and the argument field. The label field is optional, but if it is present, it must begin in column 1. The opcode field is optional, but if it is present, it must not begin in column 1. If both a label and an opcode are present, one or more spaces and/or TAB characters must separate the two. If the opcode requires arguments, they are placed in the argument field which is separated from the opcode field by one or more spaces and/or TAB characters. Finally, an optional comment can be added to the end of the line. This comment must begin with a semicolon which signals the assembler to pass the rest of the line to the listing and otherwise ignore it. Thus, the source line looks like this:

[label][opcode[arguments]][;commentary]

where the [ ] indicates that the specified item is optional.

Examples:

```
column 1
   |
   v
   GRONK   DJNZ  R1, LOOP        ; This line has everything.
           INC   R1              ; This line has no label.
   BEEP                          ; This line has no opcode.
   ; This line has no label and no opcode.

   ; The previous line has nothing at all.
           END                   ; This line has no argument.
```

Labels

A label is any sequence of alphabetic or numeric characters starting with an alphabetic. The legal alphabetics are:

! $ % & : ? [\ ] ^ _ ` { | } ~ A-Z a-z

The numeric characters are the digits 0-9. Note that "A" is not the same as "a" in a label. This can explain mysterious U (undefined label) errors occurring when a label appears to be defined.

A label is permitted on any line except a line where the opcode is IF, ELSE, or ENDIF. The label is assigned the value of the assembly program counter before any of the rest of the line is processed except when the opcode is BIT, EQU, ORG, REG, or SET.

Labels can have the same name as opcodes, but they cannot have the same name as pre-defined bit variables, operators, registers, or pre-defined byte addresses. The reserved names are:

Pre-defined bit variables:

```
AC        CY        EA        ES        ET0       ET1
EX0       EX1       F0        IE0       IE1       IT0
IT1       OV        P         PS        PT0       PT1
PX0       PX1       RS0       RS1       RB8       REN
RI        SM0       SM1       SM2       TB8       TF0
TF1       TI        TR0       TR1
```

Operators:

```
AND       EQ        GE        GT        HIGH      LE
LOW       LT        MOD       NE        NOT       OR
SHL       SHR       XOR
```

Registers:

```
A         AB        C         DPTR      PC        R0
R1        R2        R3        R4        R5        R6
R7
```

Pre-defined byte addresses:

```
$         ACC       B         DPH       DPL       IE
IP        P0        P1        P2        P3        PCON
PSW       SBUF      SCON      SP        TCON      TH0
TH1       TL0       TL1       TMOD
```

If a label is used in an expression before it is assigned a value, the label is said to be "forward-referenced." For example:

```
L1   EQU  L2 + 1   ; L2 is forward-referenced here.
L2
L3   EQU  L2 + 1   ; L2 is not forward-referenced here.
```

Numeric Constants

Numeric constants are formed according to the Intel convention. A numeric constant starts with a numeric character (0-9), continues with zero or more digits (0-9, A-F), and ends with an optional base designator. The base designators are H for hexadecimal, none or D for decimal, O or Q for octal, and B for binary. The hex digits a-f are converted to upper case by the assembler. Note that a numeric constant cannot begin with A-F as it would be indistinguishable from a label. Thus, all of the following evaluate to 255 (decimal):

```
0ffH   255   255D   377O   377Q   11111111B
```

String Constants

A string constant is zero or more characters enclosed in either single quotes (' ') or double quotes (" "). Single quotes only match single quotes, and double quotes only match double quotes, so if you want to put a single quote in a string, you can do it like this: "'". In all contexts except the DB statement, the first character or two of the string constant are all that are used. The rest is ignored. Noting that the ASCII codes for "A" and "B" are 41H and 42H, respectively, will explain the following examples:

```
"" and ''          evaluate to 0000H
"A" and 'A'        evaluate to 0041H
"AB"               evaluates to 4142H
```

Note that the null string "" is legal and evaluates to 0000H.

Expressions

An expression is made up of labels, numeric constants, and string constants glued together with arithmetic operators, logical operators, and parentheses in the usual way that algebraic expressions are made. Operators have the following fairly natural order of precedence:

Highest	anything in parentheses
	unary +, unary -
	*, /, MOD, SHL, SHR
	binary +, binary -
	LT, LE, EQ, GE, GT, NE
	NOT
	AND
	OR, XOR
Lowest	HIGH, LOW

A few notes about the various operators are in order:

1) The remainder operator MOD yields the remainder of the division of the left operand by the right operand.

2) The shifting operators SHL and SHR shift the left operand to the left or right the number of bits specified by the right operand.

3) The relational operators LT, LE, EQ, GE, GT, and NE can also be written as <, <= or =<, =, >= or =>, and <> or ><, respectively. They evaluate to 0FFFFH if the statement is true, 0 otherwise.

4) The logical operators NOT, AND, OR, and XOR perform bitwise operations on their operand(s).

5) HIGH and LOW extract the high or low byte, of an expression.

6) The special symbol $ can be used in place of a label or constant to represent the value of the program counter before any of the current line has been processed.

Some examples are in order at this point:

```
2 + 3 * 4                          evaluates to 14
(2 + 3) * 4                        evaluates to 20
NOT 11110000B XOR 00001010B        evaluates to 00000101B
HIGH 1234H SHL 1                   evaluates to 0024H
001Q EQ 0                          evaluates to 0
001Q = 2 SHR 1                     evaluates to FFFFH
```

All arithmetic is unsigned with overflow from the 16-bit word ignored. Thus:

```
32768 * 2                          evaluates to 0
```

Bit Expressions

Byte addresses are specified by expressions as described above. Bit addresses are specified by a special type of expression which can take the following three forms:

bit_variable

Bit variables are declared with the BIT pseudo-op. A number of bit variables are pre-defined by the assembler such as OV for the overflow flag (bit 2 of location 0E0H).

expression_1 . expression_2

This yields the bit address of bit expression_2 of location expression_1. expression_1 must in the range 20H - 2FH or an even multiple of 8 in the range 80H - 0FFH. expression_2 must be in the range 0 - 7. An illegal value in either expression will cause a V (value) error.

expression

An expression can also specify a bit address by itself.

■8051 OPCODES

The opcodes of the 8051 processor are divided into groups below by the type of arguments required in the argument field of the source line. If an opcode requires multiple arguments, these must be placed in the argument field in order (unless otherwise specified) and separated by commas. Some shorthand notations will be used. They are:

#imm	immediate value (e.g. #012H) in the range -128 to +255
bit	bit expression
dir	byte address in the range 0 - 0FFH
rel	byte address
Rn	R0, R1, R2, R3, R4, R5, R6, or R7

No Arguments

The following opcodes allow no arguments at all in their argument fields:

NOP RET RETI

One Argument

The following opcodes require one argument. The allowable argument values for each opcode are listed below:

OPCODE(S)	**ARGUMENT VALUE(S)**
DA, RL, RLC, RR, RRC, SWAP	A
DIV, MUL	AB
POP, PUSH	dir
DEC	A, Rn, @R0, @R1, or dir
INC	A, DPTR, Rn, @R0, @R1, or dir
SETB	C, or bit
CLR, CPL	A, C, or bit
JMP	@A + DPTR

Two Arguments in Order

The following opcodes require two arguments in the specified order:

OPCODE	FIRST ARGUMENT	SECOND ARGUMENT
MOVC	A	@A + DPTR or @A + PC
XCHD	A	@R0 or @R1
XCH	A	dir, Rn, @R0, or @R1
ADD, ADDC, ANL, ORL, SUBB, XRL	A	dir, #imm, Rn, @R0, @R1
ANL, ORL, XRL	dir	A, or #imm
ANL, ORL	C	bit, or /bit
JB, JBC, JNB	bit	rel
DJNZ	Rn, or @Rn	rel

Two Arguments in Either Order

The following opcodes require two arguments in either order:

OPCODE	FIRST ARGUMENT	SECOND ARGUMENT
MOVX	A	@DPTR, @R0, or @R1

Three Arguments in Order

The CJNE opcode requires three arguments in order. Two sets of arguments are allowed as follows:

OPCODE	FIRST ARGUMENT	SECOND ARGUMENT	THIRD ARGUMENT
CJNE	A	dir or #imm	rel
	Rn, @R0, or @R1	#imm	rel

Relative Branches

The opcodes in this group require one argument. It is an arithmetic expression that evaluates to an address in the range -128 to +127 bytes from the address of the first byte of the NEXT instruction. The opcodes are:

JC JNC JNZ JZ SJMP

Absolute Branches

These opcodes require one argument. It is an arithmetic expression that evaluates to an address on the same 2K page as the NEXT instruction. The opcodes are:

ACALL AJMP

Long Branches

The following opcodes require one argument that is an arithmetic expression with any value.

LCALL LJMP

MOV

The MOV opcode requires two arguments. The following table has the allowable combinations marked with an X:

'MOV' OPCODE ARGUMENT COMBINATIONS								
FIRST ARGUMENT	**SECOND ARGUMENT**							
	A	#imm	dir	Rn	@R0	@R1	bit	C
A		X	X	X	X	X		
dir	X	X	X	X	X	X		
Rn	X	X	X					
@R0	X	X	X					
@R1	X	X	X					
bit								X
C							X	
DPTR		*						

* = The immediate value in this case (only) is not restricted to the range -128 to +255.

■PSEUDO OPCODES

Unlike 8051 opcodes, pseudo opcodes (pseudo-ops) do not represent machine instructions. They are, rather, directives to the assembler. These directives require various numbers and types of arguments. They will be listed individually below.

BIT

The BIT pseudo-op allows the user to create bit variables. It takes the following form:

```
        label     BIT       bit_expression
```

The label is mandatory. If it is missing, the assembler will generate an L (label) error.

DB

The DB pseudo-op allows arbitrary bytes to be spliced into the object code. Its argument is a chain of zero or more expressions that evaluate to -128 through 255 separated by commas. If a comma occurs with no preceding expression, a 00H byte is spliced into the object code. The sequence of bytes 0FEH, 0FFH, 00H, 01H, 02H could be spliced into the code with the following statement:

```
        DB        -2, -1, , 1, 2
```

A special case exists here. If a string constant is entered with no arithmetic done on it, then the entire string is spliced into the code stream. Thus, the sequence of bytes 002H, 043H, 041H, 054H, 044H could be spliced into the code with the following statement:

```
        DB        1 + 1, "CAT", "C" + 1
```

DS

The DS pseudo-op is used to reserve a block of storage for program variables, or whatever. This storage is not initialized in any way, so its value at run time will usually be random. The argument expression (which may contain no forward references) is added to the assembly program counter. The following statement would reserve 10 bytes of storage called "STORAGE":

```
        STORAGE   DS        10
```

DW

The DW pseudo-op allows 16-bit words to be spliced into the object code. Its argument is a chain of zero or more expressions separated by commas. If a comma occurs with no preceding expression, a word of 0000H is spliced into the code. The word is placed into memory H:L (most significant byte of the word at the low address, least significant byte of the word at the high address) -- the opposite of standard Intel order. In other words, a DW 01234H places 12H at address “n” and 34H at address “n+1". The sequence of bytes 0FFH, 0FEH, 00H, 00H, 01H, 02H could be spliced into the code with the following statement:

```
DW      0FFFEH, , 0102H
```

END

The END pseudo-op tells the assembler that the source program is over. Any further lines in the source file are ignored and not passed on to the listing. If an argument is added to the END statement, the value of the argument will be placed in the execution address slot in the Intel hex object file. The execution address defaults to the program counter value at the point where the END was encountered. Thus, to specify that the program starts at label START, the END statement would be:

```
END     START
```

If end-of-file is encountered in the source file before an END statement is reached, the assembler will add an END statement to the listing and flag it with a * (missing statement) error.

EQU

The EQU pseudo-op is used to assign a specific value to a label, thus the label on this line is REQUIRED. Once the value is assigned, it cannot be reassigned by writing the label in column 1, by another EQU statement by a REG statement, or by a SET statement. Thus, for example, the following statement assigns the value 2 to the label TWO:

```
TWO     EQU     1 + 1
```

The expression in the argument field must contain no forward references.

IF, ELSE, ENDIF

These three pseudo-ops allow the assembler to choose whether or not to assemble certain blocks of code based on the result of an expression. Code that is not assembled is passed through to the listing but otherwise ignored by the assembler. The IF pseudo-op signals the beginning of a conditionally assembled block. It requires one argument that may contain no forward references. If the value of the argument is non-zero, the block is assembled. Otherwise, the block is ignored. The ENDIF pseudo-op signals the end of the conditionally assembled block. For example:

```
IF   EXPRESSION     ;This whole thing generates
DB   01H, 02H, 03H  ;  no code whatsoever if
ENDIF               ;  EXPRESSION is zero.
```

The ELSE pseudo-op allows the assembly of either one of two blocks, but not both. The following two sequences are equivalent:

```
                IF   EXPRESSION
                ... some stuff ...
                ELSE
                ... some more stuff ...
                ENDIF

        TEMP_LAB  SET  EXPRESSION
                IF   TEMP_LAB NE 0
                ... some stuff ...
                ENDIF
                IF   TEMP_LAB EQ 0
                ... some more stuff ...
                ENDIF
```

The pseudo-ops in this group do NOT permit labels to exist on the same line as the status of the label (ignored or not) would be ambiguous.

All IF statements (even those in ignored conditionally assembled blocks) must have corresponding ENDIF statements and all ELSE and ENDIF statements must have a corresponding IF statement.

IF blocks can be nested up to 16 levels deep before the assembler terminates with a fatal error.

INCL

The INCL pseudo-op is used to splice the contents of another file into the current file at assembly time. The name of the file to be INCLuded is specified as a normal string constant, so the following line would splice the contents of file "const.def" into the source code stream:

```
        INCL      "const.def"
```

INCLuded files may, in turn, INCLude other files until four files are open simultaneously.

ORG

The ORG pseudo-op is used to set the assembly program counter to a particular value. The expression that defines this value may contain no forward references. The default initial value of the assembly program counter is 0000H. The following statement would change the assembly program counter to 0F000H:

```
        ORG       0F000H
```

If a label is present on the same line as an ORG statement, it is assigned the new value of the assembly program counter.

PAGE

The PAGE pseudo-op always causes an immediate page ejection in the listing by inserting a form feed ('\f') character before the next line. If an argument is specified, the argument expression specifies the number of lines per page in the listing. Legal values for the expression are any number except 1 and 2. A value of 0 turns the listing pagination off. Thus, the following statement cause a page ejection and would divide the listing into 60-line pages:

```
        PAGE      60
```

REG

The REG pseudo-op functions like the EQU pseudo-op except that the argument must be a register -- i.e. R0 - R7 or another label defined with the REG pseudo-op. A label defined with REG can be used anywhere that R0 - R7 is called for. This includes forms like @LABEL. Like the EQU statement, the argument expression must contain no forward references. A label defined with the REG statement cannot be redefined by writing it in column 1, by an EQU statement, by another REG statement, or by a SET statement. The following pair of statements will assemble to a "MOV @R0, A" instruction:

```
        TEMP      REG       R0
        MOV       @TEMP, A
```

SET

The SET pseudo-op functions like the EQU pseudo-op except that the SET statement can reassign the value of a label that has already been assigned by another SET statement. Like the EQU statement, the argument expression may contain no forward references. A label defined by a SET statement cannot be redefined by writing it in column 1 or with an EQU statement. The following series of statements would set the value of label "COUNT" to 1, 2, then 3:

```
        COUNT     SET       1
        COUNT     SET       2
        COUNT     SET       3
```

TITL

The TITL pseudo-op sets the running title for the listing. The argument field is required and must be a string constant, though the null string ("") is legal. This title is printed after every page ejection in the listing, therefore, if page ejections have not been forced by the PAGE pseudo-op, the title will never be printed. The following statement would print the title "Random Bug Generator -- Ver 3.14159" at the top of every page of the listing:

```
        TITL      "Random Bug Generator -- Ver 3.14159"
```

ASSEMBLY ERRORS and WARNINGS

When a source line contains an illegal construct, the line is flagged in the listing with a single-letter code describing the error. The meaning of each code is listed below. In addition, a count of the number of lines with errors is kept and printed on the PC screen after the END statement is processed. If more than one error occurs in a given line, only the first is reported. For example, the illegal label "=$#*'(" would generate the following listing line:

```
        L  0000   FF 00 00      =$#*'(     LDA       R0
```

The line which caused the error is also output to the PC screen.

■ASSEMBLY ERRORS

* -- Illegal or Missing Statement

This error occurs when either:

1) the assembler reaches the end of the source file without seeing an END statement, or

2) an END statement is encountered in an INCLude file.

If you are "sure" that the END statement is present when the assembler thinks that it is missing, it probably is in the ignored section of an IF block. If the END statement is missing, supply it. If the END statement is in an INCLude file, delete it.

(-- Parenthesis Imbalance

For every left parenthesis, there must be a right parenthesis. Count them.

" -- Missing Quotation Mark

Strings have to begin and end with either " or '. Remember that " only matches " while ' only matches '.

B -- Branch Target Out of Bounds

The 8051 relative branch instructions will only reach addresses that are within -128 and +127 bytes of the first byte of the next instruction. The 8051 absolute branch instructions will only reach addresses on the same 2K page as the next instruction. If this error occurs, the source code will have to be rearranged to bring the branch target within range or a long branch instruction that will reach anywhere will have to be used.

D -- Illegal Digit

This error occurs if a digit greater than or equal to the base of a numeric constant is found. For example, a 2 in a binary number would cause an illegal digit error. Especially, watch for 8 or 9 in an octal number.

E -- Illegal Expression

This error occurs because of:

1) a missing expression where one is required

2) a unary operator used as a binary operator or vice-versa

3) a missing binary operator

4) a SHL or SHR count that is not 0 thru 15

I -- IF-ENDIF Imbalance

For every IF there must be a corresponding ENDIF. If this error occurs on an ELSE or ENDIF statement, the corresponding IF is missing. If this error occurs on an END statement, one or more ENDIF statements are missing.

L -- Illegal Label

This error occurs because of:

1) a non-alphabetic in column 1

2) a reserved word used as a label

3) a missing label on a BIT, EQU, REG, or SET statement

4) a label on an IF, ELSE, or ENDIF statement

M -- Multiply Defined Label

This error occurs because of:

1) a label defined in column 1 or with the EQU statement being redefined

2) a label defined by a SET statement being redefined either in column 1 or with the EQU statement

3) the value of the label changing between assembly passes

O -- Illegal Opcode

The opcode field of a source line may contain only a valid machine opcode, a valid pseudo-op, or nothing at all. Anything else causes this error.

P -- Phasing Error

This error occurs because of:

1) a forward reference in a BLK, CPU, EQU, ORG, or SET statement

2) a label disappearing between assembly passes

S -- Illegal Syntax

This error means that an argument field is scrambled. Sort the mess out and reassemble. In particular, look for use of registers that don't apply to a particular opcode, missing commas, and the like.

T -- Too Many Arguments

This error occurs if there are more items (expressions, register designators, etc.) in the argument field than the opcode or pseudo-op requires. The assembler ignores the extra items but issues this error in case something is really mangled.

U -- Undefined Label

This error occurs if a label is referenced in an expression but not defined anywhere in the source program. If you are "sure" you have defined the label, note that upper and lower case letters in labels are different. Defining "LABEL" does not define "Label."

V -- Illegal Value

This error occurs because:

1) an immediate value is not -128 thru 255, or

2) a direct address is not 0 thru 255, or

3) a bit expression contains an illegal address or bit number, or

4) a DB argument is not -128 thru 255, or

5) an INCL argument refers to a file that does not exist.

■WARNING MESSAGES

Some errors that occur during the parsing of the cross-assembler command line are non-fatal. The cross-assembler flags these with a message on the PC screen, beginning with the word "Warning." The messages are listed below:

Illegal Option Ignored

The only options that the cross-assembler knows are -l and -o. Any other command line argument beginning with - will cause this error.

-l Option Ignored -- No File Name
-o Option Ignored -- No File Name

The -l and -o options require a file name to tell the assembler where to put the listing file or object file. If this file name is missing, the option is ignored.

Extra Source File Ignored

The cross-assembler will only assemble one file at a time, so source file names after the first are ignored. To assemble a second file, invoke the assembler again.

Extra Listing File Ignored
Extra Object File Ignored

The cross-assembler will only generate one listing and object file per assembly run, so -l and -o options after the first are ignored.

■FATAL ERROR MESSAGES

Several errors that are detected during the parsing of the cross-assembler command line or during the assembly run are fatal. The cross-assembler flags these with a message on the PC screen beginning with the words "Fatal Error." The messages are explained below:

No Source File Specified

This one is self-explanatory. The assembler does not know what to assemble.

Source File Did Not Open

The assembler could not open the source file. The most likely cause is that the source file as specified on the command line does not exist. Rarely, a read error in the disk directory could cause this error.

Listing File Did Not Open
Object File Did Not Open

This error indicates either a defective listing or object file name or a full disk directory. Correct the file name or make more room on the disk.

Error Reading Source File

This error generally indicates a read error in the disk data space. Use your backup copy of the source file (You do have one, don't you?) to recreate the mangled file and reassemble.

Disk or Directory Full

This one is self-explanatory. Some more space must be found either by deleting files or by using a disk with more room on it.

File Stack Overflow

This error occurs if you exceed the INCLude file limit of four files open simultaneously.

If Stack Overflow

This error occurs if you exceed the nesting limit of 16 IF blocks.

Too Many Symbols

You have run out of memory. The space for the cross-assembler's symbol table is allocated at run-time, so the cross-assembler will use all available memory up to a limit of 64K. Please call or e-mail us (support@systronix.com) if you are having this error, we may be able to help.