

Small Assembler User's Manual
for MS/PC-DOS Systems

J. E. Hendrix
417 N. 11th Street
Oxford, MS 38655

Copyright 1988 J. E. Hendrix.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author. Printed in the United States of America. .nf

Cover Design:

Formatting:

TRADEMARKS:

CP/M(R) is a registered trade mark of Digital Research Inc.

IBM(R) is a registered trade mark of International Business Corporation.

Intel(R) is a registered trade mark of Intel Corporation.

Microsoft(R) is a registered trade mark of Microsoft Corporation.

MS-DOS(R) is a registered trade mark of Microsoft Corporation.

PC-DOS(R) is a registered trade mark of International Business Corporation.

TABLE OF CONTENTS

iii

SECTION 1	Introduction	1
1.1	The Small Assembler Package.	2
1.2	System Requirements	3
SECTION 2	Small Assembler Concepts and Facilities .	4
2.1	Source Files	5
2.2	Object Files	6
2.3	Machine Instructions	17
2.4	Assembler Directives	22
2.5	Expressions	31
2.6	The Macro Facility	36
SECTION 3	Assembling Programs	39
3.1	ASM: The Assembler	40
3.2	LINK: The Microsoft Linker	48
SECTION 4	The Small Assembler Utilities	52
4.1	AR: The Source Archive Utility	53
4.2	DUMP: The File Dump Utility	54
4.3	CMIT: The Configuration Utility	58
APPENDIX A	The Machine Instruction Table	63
APPENDIX B	Machine Instruction Formats	75

SECTION 1

Introduction

SECTION 1

Introduction

2

1.1 The Small Assembler Package

Small Assembler is a macro assembler designed for use on 8086-family computers under MS/PC-DOS. It was created as a companion for the Small C compiler. As such, it was designed to appeal to Small C users by stressing adaptability, and educational value. Program size and execution speed were considered less important than these primary goals. Therefore, like the compiler, Small Assembler was written in Small C and is being distributed in both source and object formats.

Small Assembler began life as the CP/M Small-Mac assembler and has retained that assembler's simplified macro facility and its distinctive use of C-language operators in expressions. Otherwise, it has been completely rewritten for compatibility with the Microsoft assembler which heretofore was the only translator for Small C output.

Although Small Assembler implements only a subset of the features of the Microsoft assembler, it does know about ALL of the processors from the 8086 to the 80386. It also assembles floating point instructions for the 8087, 80287, and 80387 numeric processor extensions.

Small Assembler creates Microsoft compatible object files and depends on the use of LINK (which comes with DOS) for linking its output. Although Small Assembler can generate segments larger than 64 Kbytes, OBJ files containing such segments are incompatible with LINK which cannot handle large segments.

The salient features of Small Assembler are:

1. ease of use
2. a simplified macro facility
3. the use of C-language expression operators
4. object and executable file visibility
5. an externally defined machine instruction table

The following programs are included in the Small Assembler package:

AR	source archive maintainer
ASM	macro assembler
CMIT	machine instruction table compiler
DUMP	formatted dump utility for OBJ, EXE, and other files

AR is a program that maintains source archives. Since Small Assembler sources include a large number of (mostly) small utility functions, AR was used to combine them into a single archive file (ASM.ARC) for distribution purposes. You must use AR, therefore, to extract from

ASM.ARC whatever source modules you are interested in working with.
The three major ASM source files are not in the archive.

ASM is a two-pass, table driven, relocatable, macro assembler. It
"learns" the target machine from a machine instruction table (MIT)
which is created with a text editor and compiled with the CMIT
utility.

SECTION 1

Introduction

3

CMIT compiles machine instruction tables, lists them, and generates an OBJ file which, when linked with ASM, configures it to the target CPUs. This approach to defining machine instructions to the assembler provides a great deal of flexibility in adapting the assembler to different CPUs and in creating customized instruction sets.

DUMP produces a formatted dump of OBJ, EXE, and other files. Every OBJ record is deciphered into a meaningful format. EXE file headers are likewise deciphered, the remainder of EXE files are formatted in the usual hex/ASCII way. Other files are dumped entirely in the hex/ASCII format. This program makes it easy to study the contents of OBJ files and EXE file headers.

1.2 System Requirements

This implementation of the Small Assembler package runs on 8086-family machines using the MS-DOS and PC-DOS operating systems (hereafter called DOS). Two diskette drives and 256 Kbytes of memory are sufficient to run the assembler.

SECTION 2

Small Assembler Concepts and Facilities

SECTION 2

Small Assembler Concepts and Facilities

5

2.1 Source Files

Small Assembler source files are standard ASCII files. They may be created with any pure ASCII text editor. A source line consists of zero or more characters terminated by a carriage-return, line-feed sequence. A line may contain at most 80 data characters, besides the terminating sequence.

Source lines have a free field format with fields appearing in the following order on each line:

symbol operation operand comment

Each field is optional and null lines are ignored. Fields are separated by white space (spaces and tabs) and comments are prefixed with a semicolon (;).

2.1.1 The Symbol Field

A symbol consists of a sequence of letters, digits, and the special characters "_", "\$", "?", and "@". The first character must not be a digit. Upper and lower case letters may be used. Normally these are equivalent; however, the case sensitivity switch (-C) or the .CASE directive may be used to make the assembler sensitive to case differences. Symbols have a maximum length of 30 characters.

Labels are symbols that take on the current value of the assembler's location counter and so can be used as names for memory locations. Labels appear with machine instructions and certain assembler directives that define memory. The address assigned to a label is the address of the first byte of the next instruction or data item.

Non-label symbols are used with assembler directives that define their meaning.

Labels may appear alone or followed by a machine instruction or assembler directive and/or comments. Labels that precede machine instructions are always terminated with a colon; others are not. References (which appear in the operand field) to symbols are never terminated with colons.

2.1.2 The Operation Field

Two kinds of mnemonic entry may appear in the operation field -- machine instructions (also called "operation codes" or "op-codes") and assembler directives (also called "pseudo-ops"). Op-codes cause the

assembler to assemble instances of machine instructions into the output file; whereas, directives cause it to take some other action.

If no symbol precedes an op-code or directive, then the operation field may begin in the first character position.

SECTION 2

Small Assembler Concepts and Facilities

6

NOTE: Directives are defined within the assembler, but op-codes are defined in an external machine instruction table (80X86.MIT) which is compiled into internal format and placed into an object file (80X86.OBJ) by the configuration utility CMIT. This object file is then linked with the assembler to produce a copy of ASM.EXE which knows the instructions in the machine instruction table.

2.1.3 The Operand Field

Operands and/or operand locations (memory addresses or register names) are specified in the operand field.

Symbols in the operand field must be defined elsewhere in the program.

The dollar sign "\$" may be used in the operand field as an implied label for the address of the current instruction.

A question mark "?" may be used for "don't care" values. Small Assembler assembles them with the value zero.

Memory references and numeric values may be written as expressions. Small Assembler expression operators are a subset of those in the C language and follow the same precedence and grouping rules. This gives C/assembly programmers just one set of expression evaluation rules to learn. For more about expressions see "Expressions" below.

2.1.4 Comments

Comments may appear as the last field on any line. A semicolon introduces each comment. A line may consist entirely of a comment by specifying a semicolon as the first graphic character. White space is not necessary before a comment.

2.2 Object Files

Since Small Assembler is intended for student use and comes with a dump utility for exposing the secrets of object files, the following material has been provided to make the assembler's output intelligible. See Section 4.2 for a sample dump of an object file.

Small Assembler object files implement a subset of the Microsoft MS-DOS object file format which in turn is a subset of the Intel format.

An object file is a collection of variable length records which are arranged in a prescribed sequence. Each record begins with a byte which declares the type of record. Intel has given each type of record a unique name. The records which Small Assembler generates are:

SECTION 2

Small Assembler Concepts and Facilities

7

CODE	NAME	DESCRIPTION
80	THEADR	translator's header record
96	LNAMES	list of names record
98	SEGDEF	segment definition record
99	SEG386	segment definition record for 80386
9A	GRPDEF	group definition record
90	PUBDEF	public declaration record
91	PUB386	public declaration record for 80386
8C	EXTDEF	external reference definition record
A0	LEDATA	logical enumerated data record
A1	LED386	logical enumerated data record for 80386
A2	LIDATA	logical iterated data record
A3	LID386	logical iterated data record for 80386
9C	FIXUPP	fixup record
9D	FIX386	fixup record for 80386
8A	MODEND	module end record
8B	MOD386	module end record for 80386

The order of the records in an object file is generally as follows:

```

THEADR
LNAMES...
SEGDEF...
[EXTDEF...]
[GRPDEF...]
[[PUBDEF...] LIDATA/LEDATA [FIXUPP...]]...
MODEND

```

Brackets in this list imply that the enclosed records are optional. Ellipses imply that more records (or groups of records) of the preceding type may follow. The slash in LIDATA/LEDATA indicates that a record of either type may occur there. Note that several of these record types have xxx386 equivalents which could appear in this diagram as well.

Every record has the following general structure:

record code	record length	record body	check sum
----------------	------------------	----------------	--------------

Format of an object record.

RECORD CODE is a 1-byte value which identifies the type of record as defined in the preceding list. RECORD LENGTH is a 1-word field that specifies the number of bytes in the record after the record length field. The first byte of the record length is the low-order byte. RECORD BODY represents the content of the record. This part of the record is unique to each record type. CHECKSUM, at the end, provides a means of checking the integrity of the record. It contains the two's

SECTION 2

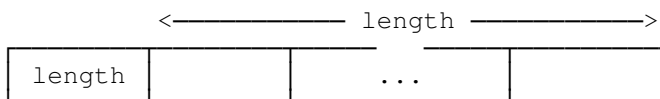
Small Assembler Concepts and Facilities

8

complement of the 8-bit (modulo 256) sum of all of the preceding bytes in the record (including the code and length fields). When reading an object record, the 8-bit sum of all of its bytes (including the checksum) should be to zero.

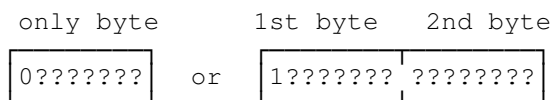
Many of the fields in the body of object records are of specific types. To eliminate redundancies, these field types are described once, before the records themselves are treated.

NAME fields are variable length strings containing the symbolic names of things like modules, segments, groups, external references, etc. The first byte of a NAME field contains a value between 0 and 127 which is the length in bytes of the string which follows immediately. Although a name may be 127 bytes long in the object file, ASM imposes a limit of 30 bytes.



Format of a NAME field.

INDEX fields are used to refer to previously occurring items in the file. This arrangement saves space in object files. For example, the LNAMEs record contains a list of names. Thereafter, instead of specifying an actual name, subsequent records refer to the first LNAME name as 1, the second as 2, etc. INDEXes may be as large as 32767, although values greater than 127 are fairly rare. So, to save space, small INDEX values occupy only one byte. If the high-order bit of the first byte of an index is zero, then it has a value between 0 and 127, and occupies only one byte. However, if it is not zero, then the remaining seven bits are the high-order part of the index and the following byte contains the low-order 8-bits. Notice that this is backwards from the normal, low-to-high order sequence of multi-byte numbers. Since indexes begin counting at one, the value zero is sometimes given special meaning.



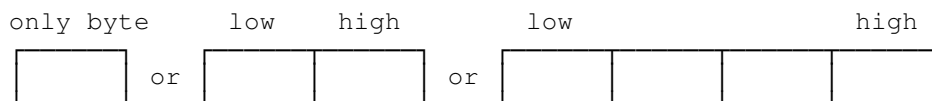
Formats of an INDEX field.

NUMBER fields are used to convey segment lengths, offsets, etc. They contain binary values of length 1, 2, or 4 bytes. Multi-byte NUMBER fields always occur with the least significant byte first. The length

of a NUMBER field depends on its context. Fields described below as segment LENGTHs, OFFSETs, or DISPLACEMENTs are either 2 or 4 bytes in length. They have 2 bytes in normal records and 4 bytes in xxx386 records.

SECTION 2 Small Assembler Concepts and Facilities

9

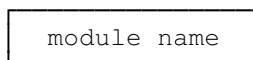


Formats of a NUMBER field.

In the descriptions which follow only the format of the body of each record type is given. It is understood that the record is prefixed with code and length fields and suffixed with a checksum. The numbers and number ranges shown above the record diagrams are the lengths in bytes of the fields beneath. An asterisk following a field length specification indicates that the field is optional. Some record fields contain specific numeric values which ASM always generates. These are fields that add to the flexibility of the record in ways that ASM does not utilize. For brevity, these fields are not explained.

2.2.1 THEADR -- Translator's Header Record

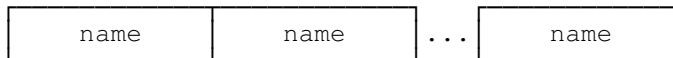
1-128



The THEADR record is the first record of an object file. It supplies the name of the object module in a single NAME field. ASM takes this name from the filename of the object file itself.

2.2.2 LNames -- List of Names Record

1-128 1-128 1-128



The LNames record contains nothing but a sequence of NAME fields. The names may be the names of segments, classes, or groups. The position of a name determines how it is referenced by INDEX fields in subsequent SEGDEF and GRPDEF records. For the first LNames record, the first name has index value 1, the second 2, and so on. For the second LNames record, the first name has the index value N+1 where N is the number of names in the first LNames record, and so on for following LNames records. The end of the record is determined by the record length field.

2.2.3 SEGDEF/SEG386 -- Segment Definition Record

SECTION 2 Small Assembler Concepts and Facilities

10

1	2*	2-4*	2-4	1-2	1-2	1-2
align combine big seg	segment frame number	segment offset	segment length	segment name index	class name index	overlay name index

A SEGDEF or SEF386 record is needed to define each memory segment in the module. The first byte encodes three segment attributes -- the alignment type, the combination type, and the "big segment" status. This byte has the following format:

align	combine	b	0
-------	---------	---	---

The ALIGN bits are encoded as follows:

SYMBOL	VALUE	DESCRIPTION
A_ABS	0	absolute segment
A_BYTE	1	byte aligned
A_WORD	2	word aligned
A_PARA	3	paragraph aligned
A_PAGE	4	page aligned

These symbolic names, and those that follow, are the ones that appear within the ASM source files. They are also used without the prefix (x_) in object file listings produced by DUMP.

If and only if the segment has absolute alignment, the SEGMENT FRAME NUMBER and SEGMENT OFFSET fields are present to specify the location of the segment in memory.

NOTE: The initial release of Small Assembler does not recognize absolute segments.

The COMBINE bits are encoded as follows:

SYMBOL	VALUE	DESCRIPTION
C_NOT	0	does not combine
C_PUBLIC	2	concatenates
C_STACK	5	concatenates

C_COMMON 6 overlaps

LINK combines segments only if they have the same names and combine types and the combine type is not C_NOT. Segments that are concatenated behind others have their addresses incremented by the total length of the preceding segments. References to these concatenated segments are adjusted as

SECTION 2

Small Assembler Concepts and Facilities

11

necessary. C_COMMON segments are combined into the same memory space; their addresses are not adjusted by LINK. The space occupied by C_COMMON segments is as large as the largest combined segment.

The BIG SEGMENT bit (b) has the following meanings:

SYMBOL	VALUE	DESCRIPTION
B_NOTBIG	0	not a big (64k) segment
B_BIG	1	is a big (64k) segment

Since a segment may be a full 64K bytes in length, but the 16-bit SEGMENT LENGTH field can only go as high as 64K-1, additional information is needed to specify a "big" (64K byte) segment length. This bit specifies that situation. The SEGMENT LENGTH field must be zero for big segments.

SEGMENT FRAME NUMBER specifies the paragraph address from which the beginning address of an absolute segment is calculated. Paragraphs fall on 16-byte boundaries. This field is present only if the segment is an absolute segment.

SEGMENT OFFSET specifies the offset from the beginning of the segment frame to the point where an absolute segment starts. This field is present only if the segment is an absolute segment.

SEGMENT LENGTH is a number field specifying the length of the segment in bytes. For SEGDEF records, this field may have values from 0 through 65535. A segment that is 65536 long is indicated by zero in this field and one in the BIG SEGMENT bit.

SEGMENT NAME INDEX points to the name of the segment as defined in a previous L NAMES record.

CLASS NAME INDEX points to the name of the segment's class as defined in a previous L NAMES record. If the segment was not given a class name, then this field points to a null (zero length) name which ASM always generates.

OVERLAY NAME INDEX always points to a null name. This field was used by versions of LINK prior to 2.40, but is no longer used.

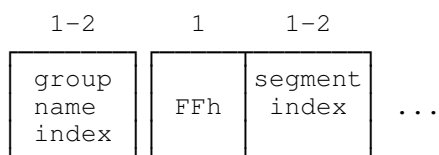
As you can see, a SEGDEF or SEG386 record defines a segment by specifying its

1. name,
2. class name,
3. length,
4. type of alignment,
5. method of combining with other segments, and
6. absolute location in memory (if necessary).

SECTION 2 Small Assembler Concepts and Facilities 12

So that they can be referenced by subsequent records, ASM assigns an index value to each segment defined by a SEGDEF or SEG386 record. The first segment is indexed by 1, the second by 2, and so on.

2.2.4 GRPDEF -- Group Definition Record

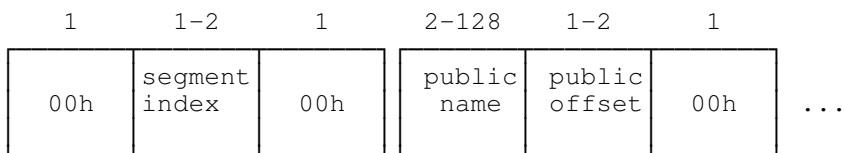


A group is a concatenated collection of segments which are addressed by the program as a single large segment. GRPDEF records define groups by naming them and identifying the constituent segments.

GROUP NAME INDEX points to the name of the group as defined in a preceding L NAMES record.

Following are one or more segment descriptors consisting of a byte containing FF hex and a SEGMENT INDEX which points to a previously defined segment. The number of segment descriptors is implied by the RECORD LENGTH.

2.2.5 PUBDEF/PUB386 -- Public Declaration Record



Each PUBDEF or PUB386 record declares one or more entry points into a given segment.

SEGMENT INDEX points to the previously defined segment which contains public labels.

PUBLIC NAME is a name field which contains the actual name of the label.

PUBLIC OFFSET is a number expressing the offset from the beginning of the segment to the public label's location; that is, the segment-relative value of the label.

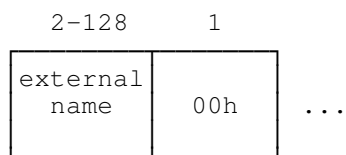
The number of public name descriptors in the record is implied by the record length.

SECTION 2

Small Assembler Concepts and Facilities

13

2.2.6 EXTDEF -- External Reference Definition Record

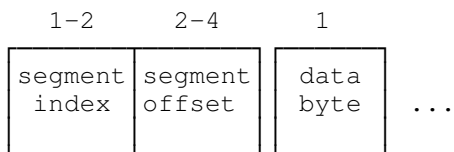


An EXTDEF record declares one or more names to be external references. LINK takes note of such names and seeks to ensure that each one matches a public name in some module linked with the program. These can be modules that are specifically named in LINK's command line, or modules that LINK finds in object libraries to which it is directed.

The body of an EXTDEF record contains one or more external name descriptors as shown above. EXTERNAL NAME is a name field containing the actual name of the external reference.

Each external name is assigned an index value for reference by subsequent FIXUPP records. The first name has index value 1, the second 2, and so on.

2.2.7 LEDATA/LED386 -- Logical Enumerated Data Record



This is one of two types of record that specify the actual data which constitutes the module. In this type of record the data is enumerated (listed) byte by byte.

SEGMENT INDEX designates a previously defined segment (SEGDEF record) to be the recipient of the data.

SEGMENT OFFSET is a number specifying the location, relative to the beginning of the segment, at which the first data byte is to be placed. Subsequent bytes are placed in consecutive locations.

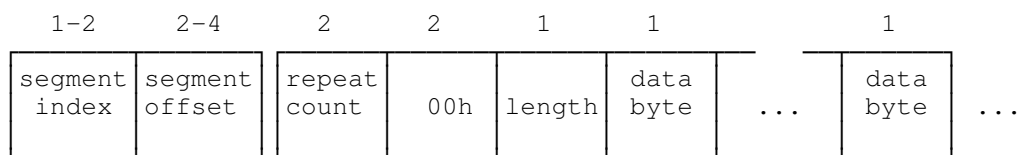
DATA BYTE is the first of up to 1024 data bytes that go into the indicated segment. The number of data bytes is implied by the record length.

2.2.8 LIDATA/LID386 -- Logical Iterated Data Record

SECTION 2

Small Assembler Concepts and Facilities

14



This second type of data record is used to convey to the linker repeated (iterated) data items. ASM generates this kind of data record when it encounters data definition statements with operands like 5 DUP(128) which calls for five occurrences of the value 128. This has obvious advantages in reducing the size of object files, especially when the repeat count is large.

SEGMENT INDEX points to the segment which is the recipient of the data.

SEGMENT OFFSET is a number that gives the offset from the beginning of the segment to the point where the data is to be located.

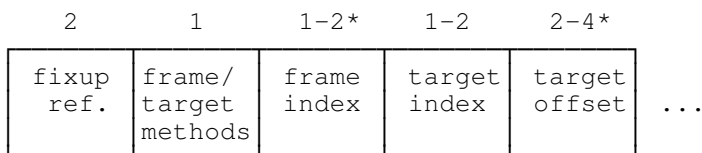
The next group of fields form a block which specifies the actual data content. More than one such block may be included in an LIDATA or LID386 record. The last block is implied by the record length.

REPEAT COUNT is a number that specifies how many iterations of the following data that are to be loaded.

LENGTH is a number that indicates the length in bytes of the data.

DATA BYTE is one of the LENGTH bytes that constitutes one iteration of the data.

2.2.9 FIXUPP/FIX386 -- Fixup Record



When LINK concatenates segments it must also adjust (or fix) references to locations in segments which do not come first in the order of concatenation. In addition, references to segment names, which imply the paragraph address of the named segments, must be fixed by the DOS loader when it determines the final resting places of the named segments. These references are embedded within the segments;

their locations must be made known to LINK so it can either patch them or pass the information through the EXE file to the DOS loader for run-time patching of segment name references.

As indicated above, FIXUPP and FIX386 records always follow a data record to which they refer. That is, the fixups in a FIXUPP or FIX386 record are applied to the data in the immediately preceding LxDATA or LxD386 record.

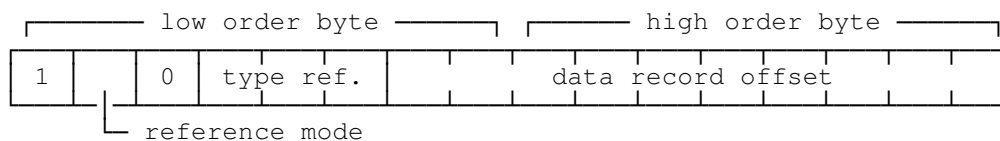
SECTION 2

Small Assembler Concepts and Facilities

15

Five fields (above) constitute a block which fully defines a single fixup. More than one such block may occur in a single FIXUPP or FIX386 record. The last block is implied by the record length.

FIXUP REFERENCE is a word length field which encodes enough information for LINK to determine the location and type of the reference to be patched. It has the following format:



The first bit (in the low byte) is always one to indicate that this block defines a "fixup" as opposed to a "thread." Since ASM does not utilize threads, they are not discussed here.

The REFERENCE MODE bit indicates how the reference is made, as follows:

SYMBOL	VALUE	DESCRIPTION
F_M_SELF	0	self-relative reference
F_M_SEGMENT	1	segment-relative reference

Self-relative references locate a target address relative to the CPU's instruction pointer (IP); that is, the target is a certain distance from the location currently indicated by IP. This sort of reference is common to the jump instructions. Such a fixup is not necessary unless the reference is to a different segment. Segment-relative references locate a target address in any segment relative to the beginning of the segment. This is just the "displacement" field that occurs in so many instructions.

The TYPE REFERENCE bits (called the LOC bits in Microsoft documentation) encode the type of reference as follows:

SYMBOL	VALUE	DESCRIPTION
F_L_LO	0	low byte of an offset
F_L_OFF	1	offset part of a pointer

F_L_BASE	2	base (segment) part of a pointer
F_L_PTR	3	pointer (offset/base pair)
F_L_HI	4	high byte of an offset

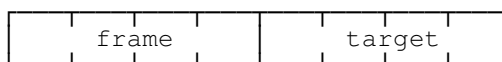
SECTION 2

Small Assembler Concepts and Facilities

16

The DATA RECORD OFFSET subfield specifies the offset, within the preceding data record, to the reference. Since a record can have at most 1024 data bytes, 10 bits are sufficient to locate any reference.

FRAME/TARGET METHODS is a byte which encodes the methods by which the fixup "frame" and "target" are to be determined, as follows:



Each reference is relative to some frame (segment) address in a segment register. LINK must know this "frame of reference" in order to properly patch the reference.

The FRAME bits tell LINK to determine the frame in one of the following ways:

SYMBOL	VALUE	DESCRIPTION
F_F_SI	0	frame given by a segment index
F_F_GI	1	frame given by a group index
F_F_EI	2	frame given by an external index
F_F_LOC	4	frame is that of the reference location
F_F_TAR	5	frame is determined by the target

In the first three cases FRAME INDEX is present in the record and contains an index of the specified type. In the last two cases there is no FRAME INDEX field.

The TARGET bits tell LINK to determine the target of the reference in one of the following ways:

SYMBOL	VALUE	DESCRIPTION
F_T_SID	0	target given by a segment index + displacement
F_T_GID	1	target given by a group index + displacement
F_T_EID	2	target given by an external index + displacement
F_T_SIO	4	target given by a segment index alone
F_T_GIO	5	target given by a group index alone
F_T_EIO	6	target given by an external index alone

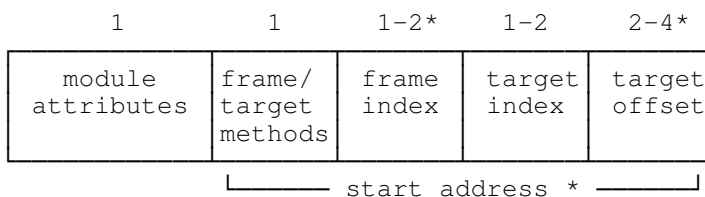
In each case TARGET INDEX is present in the record and contains an

index of the indicated type. In the first three cases TARGET OFFSET is present and specifies the offset from the location of the segment, group, or external address to the target. In the last three cases there is no TARGET OFFSET because an offset of zero is assumed.

SECTION 2 Small Assembler Concepts and Facilities

17

2.2.10 MODEND -- Module End Record



One occurrence of a MODEND record terminates each object module. The MODULE ATTRIBUTES byte is encoded as follows:

attr	0	0	0	0	0	1
------	---	---	---	---	---	---

The ATTR bits indicate whether or not the module is a "main" module; that is, whether or not it is the module of a program in which execution should begin. It also indicates whether or not a starting address is included in the MODEND record. These bits are encoded as follows:

SYMBOL	VALUE	DESCRIPTION
M_A_NN	0	non-main module, no start address
M_A_NA	1	non-main module, start address
M_A_MN	2	main module, no start address
M_A_MA	3	main module, start address

In cases 1 and 3 a START ADDRESS is specified by means of four fields that have the same semantics as corresponding fields in a FIXUPP or FIX386 record. They specify the frame in which the start address is located and its location with respect to some segment, group, or external address. In fact, ASM only uses the segment option.

2.3 Machine Instructions

As mentioned above, machine instructions are defined in an external machine instruction table (MIT). This is an ASCII file which specifies the opcode mnemonics, the operand syntax, and the object code to be generated for each machine instruction. The configuration utility CMIT compiles this table into internal format, then optionally lists it and/or generates an OBJ file for linking with the assembler.

This approach to defining the machine instructions makes it much easier to adapt this assembler to future CPUs in the 8086 family. It also functions as a very handy piece of documentation that fully describes with complete accuracy every aspect of the instruction set.

SECTION 2 Small Assembler Concepts and Facilities 18

Appendix A lists the machine instruction table for the 8086, 80186, 80286, and 80386 central processing units as well as the 8087, 80287, and 80387 numeric processor extensions (NPX). Each line in the MIT consists of five fields ordered as follows:

processor object mnemonic operand comment

White space separates the fields. A semicolon introduces each comment field.

2.3.1 The Processor Field

The first field is the processor field. This field encodes the first CPU or NPX to implement the instruction being defined. The assumption is that once implemented, an instruction continues to be supported by future processors. This is not always true of the NPX instructions, but it is a good general rule for deciding which instructions should be allowed in a program depending on the stated target processor. If this code indicates a later processor than the target, the instruction is disallowed.

The PROCESSOR field is written as a single hexadecimal byte. The first nibble carries the NPX information. It is zero for 80x86 instructions. For NPX instructions it encodes the first processor as follows:

CODE	NPX
0	8087
2	80287
3	80387

The second nibble encodes the CPU information as follows:

CODE	CPU
0	8086
1	80186
2	80286
3	80386

2.3.2 The Object Field

The object field tells the assembler how to generate output code. It consists of a string of "op-code directives" or "formatting commands" connected by underscore (_) or plus sign (+) characters. An op-code directive may be a hexadecimal byte or a symbolic code that directs the assembler to take some particular action. Hexadecimal bytes are output as is, except when they are followed by a plus sign; in that case the byte is modified according to the following directive before being generated. The symbolic op-code directives are:

DIRECTIVE	DESCRIPTION
<code>_/0#</code>	generate a ModR/M byte with the #th register/memory operand encoded as (mm000R/M)
<code>_/1#</code>	generate a ModR/M byte with the #th register/memory operand encoded as (mm001R/M)
<code>_/2#</code>	generate a ModR/M byte with the #th register/memory operand encoded as (mm010R/M)
<code>_/3#</code>	generate a ModR/M byte with the #th register/memory operand encoded as (mm011R/M)
<code>_/4#</code>	generate a ModR/M byte with the #th register/memory operand encoded as (mm100R/M)
<code>_/5#</code>	generate a ModR/M byte with the #th register/memory operand encoded as (mm101R/M)
<code>_/6#</code>	generate a ModR/M byte with the #th register/memory operand encoded as (mm110R/M)
<code>_/7#</code>	generate a ModR/M byte with the #th register/memory operand encoded as (mm111R/M)
<code>_/i#&</code>	generate a ModR/M byte and displacement with the 3 low-order bits of the 6-bit #th operand in (...iii...) and the &th operand (a register or memory reference) in (mm...R/M)
<code>_/r#</code>	generate a ModR/M byte with the #th operand (a register) in both parts of (mmrrrR/M)
<code>_/r#&</code>	generate a ModR/M byte and displacement with the #th operand (a register) in (...rrr...) and the &th operand (a register or memory reference) in (mm...R/M)
<code>_/s#&</code>	generate a ModR/M byte and displacement with the #th operand (a segment register) in (...sss...) and the &th operand (a register or memory reference) in (mm...R/M)
<code>+f</code>	add 8 to the prior byte if within far procedure

+i#	add high-order 3 bits of the 6-bit #th operand to the prior byte (must follow a hexadecimal byte)
+r#	add the register code of operand # to the prior byte (must follow a hexadecimal byte)

SECTION 2 Small Assembler Concepts and Facilities 20

<code>_i#</code>	generate an immediate value from the #th operand (a constant)
<code>_o#</code>	generate an offset (16 or 32 bit) to the #th operand (a memory reference)
<code>_p#</code>	generate a far pointer (16:16 or 16:32 bit) to the #th operand (a memory reference)
<code>_w6</code>	generate a WAIT instruction (9B) if assembling for the 8086
<code>_\$#</code>	generate a self-relative displacement to the #th operand (a memory reference)

In these descriptions the pound sign (#) and ampersand (&) stand for a digit which identifies one of the operands; 1 identifies the first operand, 2 the second, etc.

Symbols like `mm000R/M`, `mm...R/M`, `mmrrrrR/M`, `..iii...`, and `..sss...` refer to the pertinent bits in the ubiquitous ModR/M byte. When assembling 80386 instructions into segments that have the 32-bit address attribute, an additional byte, called the SIB byte, may follow the ModR/M byte. See Appendix A for a description of these bytes .

When generating an instruction, ASM reads the object field from left to right, performing the indicated operations.

2.3.3 The Mnemonic Field

The mnemonic field simply contains the mnemonic opcode for the instruction. When ASM attempts to recognize an instruction, it first performs a binary lookup in a table of unique mnemonic codes taken from this field of the machine instruction table. If that fails, no such instruction exists and an error message is issued. If it succeeds, then the instruction's operands are examined to determine if the mnemonic/operand combination is valid.

2.3.4 The Operand Field

The operand field of the machine instruction table specifies how many and what type operands are defined for each mnemonic code. This field consists of a comma-separated list of "operand type" symbols. These symbols are defined as follows:

SYMBOL	DESCRIPTION
1	literal '1'
3	literal '3'
i6	6 bit immediate operand
i8	8 bit immediate operand
i16	16 bit immediate operand

SECTION 2

Small Assembler Concepts and Facilities

21

i32	32 bit immediate operand
i1632	16 or 32 bit immediate operand
/8	8 bit register/memory operand
/16	16 bit register/memory operand
/32	32 bit register/memory operand
/1632	16 or 32 bit register/memory operand
/m	any size register/memory operand
\$8	8 bit self-relative address
\$16	16 bit self-relative address
\$32	32 bit self-relative address
\$1632	16 or 32 bit self-relative address
m	any size memory operand
m8	8 bit memory operand
m16	16 bit memory operand
m32	32 bit memory operand
m1632	16 or 32 bit memory operand
m3248	32 or 48 bit memory operand
m3264	32 or 64 bit memory operand
m48	48 bit memory operand
m64	64 bit memory operand
m80	80 bit memory operand
p32	32 bit pointer to far procedure
p48	48 bit pointer to far procedure
p3248	32 or 48 bit pointer to far procedure
r8	8 bit register
r16	16 bit register
r32	32 bit register
r1632	16 or 32 bit register
AL	AL register
AX	AX register
CL	CL register
DX	DX register
EAX	EAX register
eAX	AX or EAX register
xS	any segment register
CS	CS segment register
DS	DS segment register
ES	ES segment register
FS	FS segment register
GS	GS segment register
SS	SS segment register
CRx	CR0, CR2, or CR3
DRx	DR0, DR1, DR2, DR3, DR6, or DR7
TRx	TR6 or TR7
ST	80x87 stack register ST(0)
STx	80x87 stack register ST(x)

When ASM attempts to recognize the operand part of an instruction, it evaluates each operand supplied with the instruction and assigns it one of these types. When more than one type is possible, it assigns the most specific type. It then performs a binary search on a table of unique operand type combinations taken from this field of the machine instruction table. If that fails then, one-by-one, the instruction's operand types are mapped to the next more general type and another search is performed. If necessary, all permutations of operand types

SECTION 2

Small Assembler Concepts and Facilities

22

that follow from the instruction's actual operands are tried. If all of these fail, then no such instruction exists and an error message is issued. If one of the searches succeeds, then a final search is performed to verify that the instruction's mnemonic/operand combination is valid. If this is successful, the instruction is assembled; if it fails, an error message is issued since, although the mnemonic code is valid and the operands are valid, the two do not go together.

2.3.5 The Comment Field

Like the comment field in an assembler source file, the comment field in the machine instruction table is the last field and is introduced by a semicolon. This field is used to provide a one-line description of the instruction. With this final bit of information, the machine instruction table becomes a single source for everything you ever wanted to know about every instruction known to the 8086 family of processors.

2.4 Assembler Directives

Small Assembler supports the following assembler directives:

DIRECTIVE	FUNCTION
.CASE	makes symbols case sensitive
.186	allow 8086 through 80186 CPU instructions
.286	allow 8086 through 80286 CPU instructions
.287	allow 8087 through 80287 NPX instructions
.386	allow 8086 through 80386 CPU instructions
.387	allow 8087 through 80387 NPX instructions
=	sets/resets a name to a constant expression
EQU	equates a name to a constant expression
EQU	equates an alias to a symbol
SEGMENT	starts a segment and defines its attributes
ENDS	ends a segment
GROUP	combines segments into a named group
PROC	starts a procedure
ENDP	ends a procedure
ASSUME	tells what to assume is in the segment registers
PUBLIC	declares entry point symbols
EXTRN	declares external reference symbols
DB	defines data bytes
DW	defines data words
DD	defines data double words

DF	defines data far words
DQ	defines data quad words
DT	defines data ten-byte values
ORG	sets the current segment's location counter
MACRO	begins a macro definition
ENDM	ends a macro definition
macroname	calls (expands) the named macro
END	ends the source file

In the following descriptions of assembler directives, square brackets enclose optional elements. Expressions are described in section 2.5. Character strings are enclosed in either apostrophes (') or quotes ("). If an occurrence of the delimiter is desired within the string, code two successive delimiters.

2.4.1 Case Sensitivity

.CASE

This directive causes the assembler to cease converting letters that appear in labels to upper case. The effect is to cause the upper and lower case versions of a letter to be seen as different characters. Also lower case letters in public and external names go into the OBJ file without conversion to upper case. Segment, group, and class names always go into the object file in upper case, however.

2.4.2 Set Hardware Limitation

.186
.286
.287
.386
.387

These directives tell ASM which CPU or NPX is the latest one that will be executing the current program. By default ASM assumes that the program will be running on an 8086 machine with an 8087 numeric processor. None of the instructions that were first defined for use in later processors can be used in the program. By placing one or two of these directives at the front of the source file, you can cause ASM to also recognize instructions for later processors. For example, the directive .286 causes ASM to assemble only instructions known to the 8086, 80186, and 80286 CPUs and the 8087 NPXs. The directives .386 and .287 would cause ASM to assemble instructions for the 8086, 80186, 80286, and 80386 CPUs and the 8087, and 80287 NPXs.

These directives should be used only once in a program, at the beginning.

The .386 directive is a prerequisite for the USE16 and USE32 parameters of the SEGMENT directive. Exceptions evoke the "Bad Symbol" message. The .386 directive is also a prerequisite for the OSO and ASO instruction prefixes. Exceptions evoke the "Wrong Hardware" message.

2.4.3 Set or Reset a Name to a Constant Value

name = constantexpression

SECTION 2

Small Assembler Concepts and Facilities

24

This directive sets a name to the value of a constant expression. The same symbol may be reset later by other = directives. Once a value is given to a name, the name may be used in expressions in place of the value. The value of such a name is the value last assigned to it.

2.4.4 Equate a Name to a Constant Value

```
name      EQU      constantexpression
```

This directive equates a name to the value of a constant expression. The value of the name may not be changed later with another EQU or = directive. Once a value is given to such a name, the name may be used in expressions in place of the value.

2.4.5 Equate an Alias to a Symbol

```
alias     EQU      symbol
```

This form of the EQU directive assigns a second name, an alias, to a defined symbol. The original symbol, which may be a label or a name defined by any other directive, conveys all of its attributes its alias. Thereafter, the symbol may be referred to by either of its names.

2.4.6 Define a Segment

```
name      SEGMENT  [sizes] [align] [combine] [class]
...
name      ENDS
```

These two directives delimit memory segments. They are required in every program since ASM will not assemble code or data without first knowing the segment into which it goes. The name of the segment must appear in both directives. ASM does not support nested segment definitions. This makes the name in the ENDS directive superfluous, but it is retained for compatibility with the Microsoft assembler.

Once a segment has been started, it may be exited and then reentered later. In such cases, the segment attributes should be specified with the first SEGMENT directive. Subsequent SEGMENT directives for the same segment need not specify the attributes again. In any case, the attributes that go into the OBJ file will be the last ones given. If an attribute is not given, then a default value, or the previous value, is retained.

Small Assembler will accept segment attributes in any order.

The "sizes" (or "use") attribute tells whether the segment uses, by default, 16 or 32 bit addresses and operands. This field is allowed only for 80386 programs. Its values may be USE16 or USE32. Since the default size attribute of a segment is 16 bits for non-80386 programs and 32-bits for 80386 programs, the USE32 is never needed; it serves only to make the default obvious.

SECTION 2

Small Assembler Concepts and Facilities

25

NOTE: The address size override prefix (ASO) may be used to reverse the segment default address size for an instruction. Likewise, the operand size override prefix (OSO) may be used to reverse the segment default operand size for an instruction. These mnemonics were chosen arbitrarily, since Microsoft does not currently support explicit size overrides.

The "align" attribute refers to the alignment of the segment in memory. It may be specified as any one of BYTE, WORD, PARA, or PAGE. These specify the type of boundary on which the segment is to be aligned. PARAgaph boundaries occur every 16 bytes, and PAGE boundaries occur every 256 bytes. The default is PARA.

The "combine" attribute refers to the way in which the segment is supposed to combine with other segments. The options are PUBLIC, STACK, and COMMON. PUBLIC is the default, and means that the segment will be concatenated with other segments bearing the same name, class, and combine type.

The "class" attribute is an arbitrary class name assigned to the segment. It has the form

'name'

where the apostrophes are required. The only purpose of this attribute is to further qualify the segment name in determining which segments will be combined by the linker. If the class is omitted a null class name is assumed. Class names must be unique, they cannot be the same as any other defined name.

2.4.7 Combine Segments into a Named Group

```
name      GROUP      segmentname,,,
```

This directive causes the linker to group the listed segments together under the indicated name. The segments listed in the operand field must have been defined earlier in the program.

2.4.8 Define a Procedure

```
name      PROC      [distance]
          ...
name      ENDP
```

These directives delimit a procedure (or subroutine) and designate it as a "near" or "far" procedure. The "distance" field may specify NEAR or FAR. If neither is given, NEAR is assumed. The only functional difference between a near and a far procedure is that when a CALL instruction refers to a label in a FAR procedure, a far call is generated and when a RET instruction is encountered within the range of a FAR procedure, a far return is generated; otherwise, near

SECTION 2 Small Assembler Concepts and Facilities 26

instructions are generated.

2.4.9 Tell What to Assume is in the Segment Registers

 ASSUME register:name,,,

This directive informs ASM as to which segments the individual segment registers point. ASM needs this information in order to ensure that every operand reference can be made because the operand's segment address is in a segment register. Furthermore, if the default segment register (DS for data references, SS for stack references, and CS for calls) does not point to the target segment, ASM must know which segment register prefix to generate so as to specify explicitly which segment register to base the reference upon.

The ASSUME directive may be written at any point in a program. The assumptions it establishes stick until another ASSUME directive is reached. At that point, only the indicated assumptions are changed; others remain as they were established previously.

An ASSUME directive may specify the segments or groups pointed to by any or all of the segment registers. If a register is not named in a particular ASSUME directive, its assumption remains unchanged.

The operand field "register" stands for any of the segment register names CS, SS, DS, ES, FS, or GS. "Name" is the name of a segment or a group which has been previously defined.

The keyword NOTHING may be used in place of a segment/group name in order to cause ASM to forget an assumption. Likewise the entire operand field of the directive may contain the keyword NOTHING in order to cause ASM to forget all assumptions.

It is important to realize that the ASSUME directive does not itself place any value in the segment registers. That is the programmer's responsibility. This directive only tells the assembler what to assume is in the registers.

Note that books on the subject of DOS programming explain how the segment registers are setup by the DOS loader upon program entry.

2.4.10 Declare Entry Point Symbols

 PUBLIC symbol,,,

The PUBLIC directive declares the listed symbols to be the names of

labels which can be reached from other modules. That is, the listed labels are to be known publicly. Symbols declared to be public must be defined somewhere within the program module. As we have seen, this information is output in PUBDEF records in the object file.

SECTION 2 Small Assembler Concepts and Facilities

27

2.4.11 Declare External Reference Symbols

```
EXTRN    symbol:type,,,
```

This directive declares one or more symbols to be externally defined. That is, the symbols listed are referenced in the present module but are defined in other modules (in which they are declared to be public).

Symbols which are named in an EXTRN directive must not be defined elsewhere in the current program module. The operand field "symbol" contains the name of a symbol to be declared external. "Type" declares the type of the external item since, without access to its definition, ASM would have no way of knowing what the item is. This field must contain one of the keywords BYTE, WORD, DWORD, FWORD, QWORD, TBYTE, NEAR, or FAR.

One of LINK's primary duties is to resolve external references; that is, to find matching entry points (public symbols) and to modify the external references so that they correctly address them. Besides listing a specific set of object files that are to be joined into a single program, we can also direct LINK to object libraries. In such cases, unresolved external references cause a library search for modules which can satisfy them. Declaring a symbol to be external is sufficient to cause a module containing its public definition to be loaded with the program. It need not actually be referenced in the program.

2.4.12 Define Data

```
[label] DB    string or expression,,,
[label] DW    string or expression,,,
[label] DD    string or expression or real,,,
[label] DF    string or expression,,,
[label] DQ    string or expression or real,,,
[label] DT    string or expression or real or packed,,,
```

These directives define data items in memory. If a label is present, it takes on the address of the first byte. For each size item there is a different mnemonic code as follows:

CODE	SIZE	DESCRIPTION
DB	1-byte	Bytes
DW	2-bytes	Words
DD	4-bytes	Double Words

DF	6-bytes	Far Words
DQ	8-bytes	Quadruple Words
DT	10-bytes	Ten Bytes

SECTION 2

Small Assembler Concepts and Facilities

28

Each directive is followed by a comma-separated list of values. Each value occupies one or more pieces of memory of the indicated size. The types of values that are acceptable for each directive are indicated in the format statements above.

Values may have any of a number of forms. One possibility is a string. Strings are enclosed in single or double quotes. If an occurrence of the delimiting quote is needed within a string, it can be obtained by coding two consecutive quotes. For each character in the string an item of the indicated size is generated with the character in the low-order byte of the item.

If the value is not a string, it may be written as

n DUP(...)

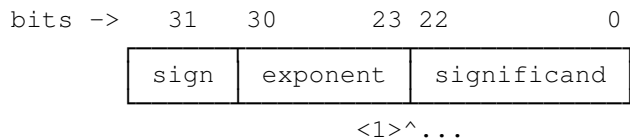
where ... is the actual value. In such cases, n occurrences of the value are generated. Non-string values may be expressions (Section 2.5), real constants, or packed decimal integers.

If the value contains a decimal point and the op-code is DD, DQ, or DT then ASM interprets the value as a real constant of the following form:

[sign] [digits] . [digits] [E [sign] digits]

White space in this description is used for clarity only; it must not be present within the number itself. Fields enclosed in square brackets are optional. The letter "E" may be written in lower case. Examples of valid real constants are .5, -1.23e+6, and 23.5e-3.

ASM converts real constants to one of three internal floating point formats which are supported by the 8087 family of numeric processors. The DD directive yields a SINGLE precision floating point number of the following form:



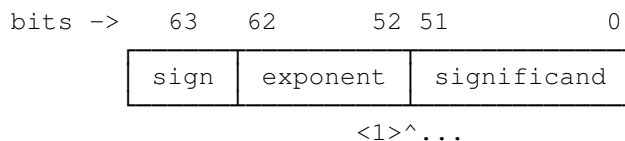
The sign bit is a one for negative values. The 8-bit exponent carries a bias of 7F hex. The significand consists of one integer bit and 23 fractional bits. Since real constants are normalized, the integer bit must be a one if the number is not zero. Since its value is known, the integer bit (shown as <1> above) is squeezed out of the significand, leaving only the fractional bits (... above). The binary point (^) is

assumed to be to the left of the remaining significand bits. A zero value contains zeroes in all fields.

The DQ directive yields a DOUBLE precision floating point number of the following form:

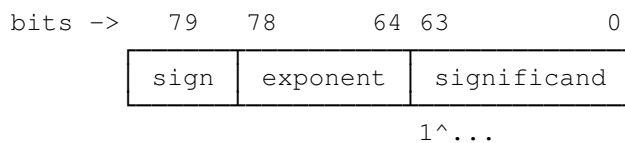
SECTION 2 Small Assembler Concepts and Facilities

29



Except for the field sizes and the exponent bias (3FF hex), double precision reals are formatted exactly like single precision reals.

The DT directive yields a TEMPORARY floating point number of the following form:

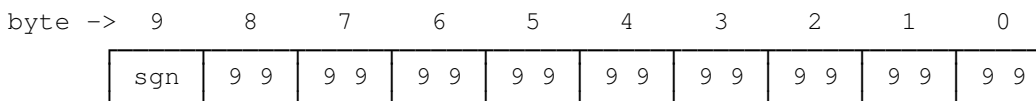


The exponent of a temporary real is biased with 3FFF hex. The significand differs from those of single and double precision reals in that the integer bit is explicit and the implied binary point falls between it and the first fractional bit (the second significand bit).

If the value does not contain a decimal point and does not have a suffix to indicate the base (b, o, d, h) and the op-code is DT then ASM interprets the value to be a decimal integer to be encoded as a PACKED decimal number. Packed values have the source format:

[sign]digits

The internal format of a packed decimal number is:



The enclosed nines each represent a binary coded decimal digit. The sign byte contains 00 if the number is positive and 80 hex if it is negative. The least significant byte, byte 0, is the one with the lowest address in memory. As with the reals, this format is compatible with the 8087 family numeric processors.

If a value is not determined, as described above, to be real or packed then it is assumed to be an expression. Except for far pointers in USE32 segments, expressions yield 32-bit quantities. If the target field is smaller than 32 bits, the low-order bits are taken. If it is larger, the value is right-adjusted in the target field.

2.4.13 Set the Current Segment's Location Counter

[name] ORG constant expression

SECTION 2

Small Assembler Concepts and Facilities

30

This directive sets a new origin (starting value) for the current segment's location counter. The location counter (assembly address) is tracked separately for each segment. When a new segment is started (by the first SEGMENT directive for the segment) its location counter is set to zero. When a segment is terminated (ENDS) the location counter is saved in the segment's symbol table entry. When an old segment is reentered, its saved value is restored to the active location counter for further use. At the end of the program, each segment's value reflects the length of the segment.

At any point within a segment an ORG directive may be used to adjust the location counter. However, since ASM will use the final value of the location counter to specify the segment's length to the object file, care must be taken not to allow it to end at some point before the actual end of the segment.

2.4.14 Define Macro

```
name      MACRO
          ...
          ENDM
```

These directives delimit a macro definition. "Name" is required to specify the name of the macro. For more on macro processing, see "The Macro Facility" below. Since Small Assembler macro parameters are identified strictly by position, formal parameter names are not needed in the operand field of the MACRO directive. Macro definitions cannot be nested.

2.4.15 Call (Expand) a Macro

```
[label]  macroname [par1[,par2,...]]
```

This directive is used to call (or expand) a macro. The label is optional. If given, it assumes the address of the first byte of the macro expansion. "Macroname" is the name given in the macro definition.

Actual parameters are supplied in a comma separated list in the operand field. Parameters are merely character sequences which replace corresponding substitution sentinels in the macro body. If spaces, commas, or semicolons are in a parameter, that parameter must be delimited with quotes (") or apostrophes ('). Delimiters within such strings are written as two successive delimiters. Missing parameters are taken as null strings. Two successive commas indicate a missing parameter. A parameter is also missing if the parameter list

terminates before the parameter's position is reached.

2.4.16 End the Source File

SECTION 2 Small Assembler Concepts and Facilities 31

[label] END [expression]

This directive designates the end of a source file. It is required and it must be the last line of each source file. If a label is present, it assumes the address of the byte following the last byte assembled. If an expression is given, it must evaluate to an address which the assembler will take as the starting address of the program. Only one starting address should be specified when a program is assembled from more than one module. If no starting address is given, execution will begin at the first instruction of the first code segment.

2.5 Expressions

Expressions may appear in the operand field of certain machine instructions or directives. For the proper placement of expressions in machine instructions, see the machine instruction table (Appendix A). Expression evaluation always produces a 32-bit binary value. If the instruction or directive requires less than 32 bits, high order bits are truncated.

Note that real and packed decimal constants may not occur in expressions. They are legal only in the data definition directives described above.

2.5.1 Relocation Rules

The value of an expression will be either absolute or segment-relative (hereafter simply called "relative") depending on whether and how symbols are used. Absolute values remain fixed through the linking process. Relative values refer to program locations which cannot be given final values until LINK has combined program segments or (in case of segment and group references) the DOS loader has placed the program in memory at its actual execution location.

A label always represents a relative value; that is, it assumes the segment relative offset (displacement) to the next assembled item in the program. Segment and group names also yield relative values. On the other hand, numeric constants are absolute.

Every expression yields a single value which itself is either relative or absolute (its relocation attribute) depending on the attributes of its primary terms and the ways in which they are combined. The following list illustrates the rules for determining the relocation attribute of an expression:

COMBINATION		RESULT
abs ?	abs	abs
abs +	rel	rel
abs ?	rel	error
rel +	abs	rel
rel -	abs	rel
rel -	rel	abs

SECTION 2

Small Assembler Concepts and Facilities

32

rel == rel	abs
rel < rel	abs
rel <= rel	abs
rel != rel	abs
rel > rel	abs
rel >= rel	abs
rel ? rel	error

A question mark in this list stands for any operator other than the ones explicitly shown for the same combination of left and right hand attributes.

A relocatable expression may generate a self-relative field; that is, a field containing a signed offset which the CPU adds to the instruction pointer to obtain the effective address. The jump instructions use self-relative addressing. In such cases ASM takes the expression as the target address. So it subtracts the location counter and the instruction length from the expression, converting it an offset from the following instruction, as the CPU requires.

2.5.2 Numbers

Numbers in expressions must be integer values. They are assumed to be decimal unless their base is explicitly indicated by one of the following suffixes:

SUFFIX	BASE
B or b	binary
O or o	octal
Q or q	octal
D or d	decimal
H or h	hexadecimal

The first character of a number must be a decimal digit. A leading zero may be needed to make hexadecimal numbers conform to this rule. Numbers are converted to 32-bit values and then combined with the rest of the expression (if any).

2.5.3 Symbols

Symbols in an expression must be defined elsewhere. They are either absolute or relative depending on how they are defined. Labels and

their aliases (defined with EQU directives) are relative, as are segment and group names. External references (declared with EXTRN directives) are also relative. Other symbols (defined with the EQU and = directives) are either absolute or relative depending on the relocation attribute of the defining expression.

SECTION 2

Small Assembler Concepts and Facilities

33

2.5.4 Operators

Most Small Assembler expression operators are a subset of those in the C language and follow the same precedence and grouping rules. For compatibility with the Microsoft assembler, several of the operators have two or three letter synonyms. Grouped by precedence, the Small Assembler operators are:

!	logical NOT	<-
~ or NOT	one's complement	<-
-	unary minus	<-
<hr/>		
*	multiplication	->
/	division	->
% or MOD	modulo (remainder)	->
<hr/>		
+	addition	->
-	subtraction	->
<hr/>		
<<	shift left	->
>>	shift right	->
<hr/>		
< or LT	less than	->
<= or LE	less than or equal	->
> or GT	greater than	->
>= or GE	greater than or equal	->
<hr/>		
== or EQ	equal	->
!= or NE	not equal	->
<hr/>		
& or AND	bitwise AND	->
<hr/>		
^ or XOR	bitwise exclusive OR	->
<hr/>		
or OR	bitwise inclusive OR	->
<hr/>		
&&	logical AND	->
<hr/>		
	logical OR	->

Operators with the highest precedence are at the top and all operators in the same box have the same precedence. Arrows indicate the direction of grouping. Parentheses may be used to control grouping. Any number of nesting levels is permitted.

In addition, the following special operators are supported:

OPERATOR	OPERATION
OFFSET ...	yield the offset part of memory reference ...
SEG ...	yield the segment part of memory reference ...
BYTE PTR ...	coerce ... to a byte reference
WORD PTR ...	coerce ... to a word reference
DWORD PTR ...	coerce ... to a double word reference
FWORD PTR ...	coerce ... to a far word reference
QWORD PTR ...	coerce ... to a quad word reference
TBYTE PTR ...	coerce ... to a ten-byte reference
NEAR PTR ...	coerce ... to a near reference
FAR PTR ...	coerce ... to a far reference
SHORT ...	coerce ... to a short reference

These operators have the same precedence and grouping as the highest precedence operators shown in the previous table.

2.5.5 The Current Instruction Address

The dollar sign "\$" may be used as an implied label for the address of the current instruction. It has the relative attribute.

2.5.6 Registers

An expression may be just a register name. For example, the instruction `MOV AX,BX` has two operand expressions both of which are register names.

2.5.7 Segment Register Overrides

If any expression in the operand field of an instruction is preceded by the name of a segment register and a colon then the indicated segment register prefix byte will be generated in front of the instruction (unless it happens to be the default for the instruction). For example,

`MOV AX,ES:data`

will generate an ES prefix byte in front of the instruction.

2.5.8 Memory References

The 8086 family processors have a very flexible memory addressing facility. Besides the segment address component (which is in a default or specified segment register), addresses may be composed of three types of segment displacement components -- (1) constant displacement, (2) relative displacement, and (3) register value. Any number of the first two may be combined according to "Relocation Rules" (Section 2.5.1). The combination of registers that are allowed depends on the CPU (8086-80286 or 80386).

SECTION 2

Small Assembler Concepts and Facilities

35

The possible combinations of address component types and the resulting types of reference are as follows:

const	label	[reg]	result
no	no	yes	indirect with no displacement
no	yes	no	direct with relocatable displacement
no	yes	yes	indirect with relocatable displacement
yes	no	no	immediate operand
yes	no	yes	indirect with absolute displacement
yes	yes	no	direct with relocatable displacement
yes	yes	yes	indirect with relocatable displacement

For CPUs in the range 8086 through 80386, operands can be addressed by the base register (BX), the base pointer (BP), the source index register (SI), or the destination index register (DI).

For example, to fetch the word pointed to by BX we would write

```
MOV AX,[BX]
```

The square brackets can be interpreted to have the meaning "contents of the memory location pointed to by." Each of these registers is associated with a default segment register from which the reference is based. They are:

INDIRECT REGISTER	SEGMENT REGISTER
BX	DS
BP	SS
SI	DS
DI	ES

In addition, displacements may be added to or subtracted from the address in the indirect register. For instance, we could write

```
MOV AX,[BX+100h]
```

Other ways this could be written are

```
MOV AX,[BX].100h  
MOV AX,[BX]+100h
```

MOV AX,100h[BX]

Finally, either of SI or DI can be added to or subtracted from either of BX or BP. And, in these cases, displacements may also be included. Therefore, forms like

SECTION 2

Small Assembler Concepts and Facilities

36

```
MOV AX,[BX+SI]
MOV AX,[BX+SI+data]
```

are also legal. (Note that "data" is a relative displacement.) These may also be written as

```
MOV AX,[BX][SI]
MOV AX,data[BX][SI]
```

In general, the valid forms of 8086 indirect reference are

[b]			
[i]			
[b+i]	[b][i]		
[b-i]			
[b+d]	[b]+d	[b].d	d[b]
[b-d]	[b]-d		
[i+d]	[i]+d	[i].d	d[i]
[i-d]	[i]-d		
[b+i+d]	d[b][i]		
[b-i-d]			

where "b" is a base register (BX or BP), "i" is an index register (SI or DI), and "d" is a constant displacement. All forms on the same line are equivalent.

The 80386 CPU uses extended, 32-bit versions of these registers and allows the use of nearly all of the CPU's 32-bit registers for indirect addressing. Any register except for ESP can serve as an index register. The only difference between an index register and another indirect addressing register is that the index register can be scaled (multiplied) by 1, 2, 4, or 8.

To specify a scaled index register append the multiplication symbol (*) followed by one of the values 1, 2, 4, or 8. If more than one register is given and neither is scaled, the second one is taken as the index. Thus

```
[eax+edx*2]
```

specifies an indirect address composed of EAX plus twice EDX. And

```
[ebx+ebx]
```

specifies EBX plus EBX. ESP cannot serve as an index register, but it can form the basis of an indirect address. See Appendix B for all possible combinations of 80386 indirect address components. Any number

of constant displacements (dis8, dis16, dis32) can be added to and/or subtracted from any legal combination of registers and one label.

2.6 The Macro Facility

SECTION 2

Small Assembler Concepts and Facilities

37

Source lines which are located between the MACRO and ENDM directives constitute the definition (or body) of a macro. They are stashed away in a buffer when they are needed. After that, the presence of a macro name in the operation field is recognized as a macro "expansion" or "call." Expanding a macro causes the entire body of the named macro to be inserted in the program as in-line code.

Neither macro definitions or macro calls may be nested.

If more than one macro definition has the same name, only the first one is used.

2.6.1 Parameter Substitution

Parameters may be specified with each macro call to tailor the expanded code to the circumstances which are unique to the call. Simply place ?1, in the definition of the macro, where the first actual parameter should go, ?2 for the second and ?0 for the 10th one. At most, ten parameters are allowed.

Parameters in a macro call are written in the operand field separated by commas. Successive commas or missing trailing parameters produce a null substitution; that is, the substitution sentinel (e.g. ?3) is squeezed out of the expanded text. If a ? is desired in the expanded text, code ?? in the macro definition. Quotes or apostrophes may delimit an actual parameter containing white space, commas, or semicolons. An occurrence of the delimiter within the string is achieved by writing two successive delimiters.

Parameter substitution is performed without regard to context. Therefore, substitutions may occur within quoted strings, comments, and even symbols and mnemonics. This simple concept can be used to great advantage.

2.6.2 Local Labels

Ten labels which are local to each macro expansion may be specified in the macro definition as @0 through @9. The first such label (and its references) encountered by the assembler will appear in the expanded text as @1, the second as @2, etc. This sequence continues throughout the program (across macro calls) so that every local label is guaranteed to be unique. This avoids "redundant definition" errors when the same macro is called repeatedly.

2.6.3 Sample Macro

The following code defines a simple macro:

```
abc      macro
          push    AX
@1:      mov     AX,?2
          call    ?1
          jc      @1
          pop     AX
          endm
```

In this case two parameters are expected with each expansion -- the name of a subroutine and a value to be passed to it. If the carry flag is set on return from the subroutine, another call is made. If this macro is expanded twice with the following lines

```
lab1:    abc      routine1,data
lab2:    abc      routine2,0100h
```

the result would be

```
lab1:    push     AX
@1:      mov     AX,data
          call    routine1
          jc      @1
          pop     AX

lab2:    push     AX
@2:      mov     AX,0100h
          call    routine2
          jc      @2
          pop     AX
```

SECTION 3
Assembling Programs

SECTION 3

Assembling Programs

40

3.1 ASM: The Assembler

Usage: ASM source [object] [-C] [-L] [-NM] [-P] [-S#]

source Source file to be assembled.

object Object file to be created.

-C Request case sensitivity.

-L Request an assembly listing.

-NM Request no macro processing.

-P Request pause on errors.

-S# Set symbol table for # symbols maximum.

3.1.1 Description

Command-line arguments may be given in any order. Switches may be introduced by a hyphen or a slash. Brackets indicate optional arguments; do not code the brackets themselves.

If an illegal switch is given, the assembler aborts after displaying the usage message shown above. The null switch (- or /) can be used to force this message when you need a reminder.

ASM returns the following exit codes to DOS for testing in batch files:

CODE	MEANING
0	the program assembled normally
1	ASM requires more memory
2	the run was canceled with a control-C
10	the program contained errors

Pressing control-S makes the assembler pause until the ENTER key is pressed. Pressing control-C aborts the run.

3.1.1.1 Source File

One source file must be specified in the command line. A drive specifier and/or path may be included with the source filename to direct the assembler to a specific drive and/or directory. If no drive or path is specified, the DOS defaults are assumed. If a source file cannot be found, ASM will abort with an error message. The default and only allowed source file name extension is ASM.

SECTION 3

Assembling Programs

41

3.1.1.2 The Object File

One object file may be specified. If none is given, the object code will go into a file on the same drive, in the same directory, and with the same name as the source file, but with a OBJ extension. A drive specifier and path may be included with the object file to direct the output to a specific drive and/or directory. If no drive and/or path is specified, the DOS defaults are assumed. The object filename must have a OBJ extension in order to be distinguished from the source file. The module name in the object file is taken from the object filename.

3.1.1.3 Case Sensitivity

Without the -C switch, all symbols are converted to upper case. With this switch, however, they are taken as is; upper and lower case variants of the same letter are considered to be different. With or without this switch, segment, group, and class names are always converted to upper case in the output OBJ file.

NOTE: The .CASE directive may be included at the beginning of the source file to produce the same effect as this switch.

3.1.1.4 Assembly Listing

An assembly listing will be produced only if the -L switch is included in the command line. The listing is sent to the standard output file and so goes to the screen unless it is redirected (e.g., >prn or >abc.lst).

Error messages always appear on the screen. If a listing is requested, they appear there also. A single source line may produce several error messages which are generated beneath the line.

The listing is paginated for 11 inch tall pages. Either a wide printer or compressed print mode should be used for assembly listings since commented source lines are almost certain to exceed 80 columns.

Each line in the listing contains (left to right):

1. the source line number in decimal.
2. the current location counter value in hexadecimal.
3. the hexadecimal object code generated by the current source line.
4. the source line.

Although the 8086 family processors store multi-byte items with the lowest order byte first, the listing shows them in the (visually) customary high-to-low order sequence for improved readability. The only exception is that far pointers show the offset part before the segment part, which is the order in which they are actually stored. Nevertheless, the offset part is in the customary high-to-low order. Segment addresses are always shown as "----" to emphasize the fact

SECTION 3

Assembling Programs

42

that their values are unknown until established by the linker or the DOS loader.

Relocatable items in the object column are flagged with an alphabetic suffix. The letter "e" flags external references; whereas, "r" flags local references. Other items are absolute; that is, they are not modified by the linker or DOS loader. If the object code will not fit in the space allotted for object code in the listing, overflow lines are printed as needed.

Values that result from EQU and = expressions are shown in the object code field even though they are not generated in the object file. To emphasize this anomaly, no location counter value appears on such lines. Also an equal sign "=" suffixes these values.

A sorted dump of the symbol table is produced at the end of the listing. Each line shows the symbol's attributes, size, value, and name. Since symbols may have multiple attributes, a column is reserved for each attribute. The attribute itself is abbreviated as:

ABBREVIATION	MEANING
=	defined by an = directive
equ	defined by an EQU directive
seg	segment name
grp	group name
cls	segment class name
pub	group name
ext	external reference name
far	far reference
pro	procedure name
cod	code label
dat	data label

The value of a segment is its length. The symbol "?" is always defined to have the value zero. This is ASM's way of allowing that special symbol in expressions.

3.1.1.5 No Macro Processing

The -NM switch, meaning "no macros," may be specified if macro processing is not required. This speeds up the assembler very slightly. Macro processing is NOT needed for programs generated by the Small C compiler.

3.1.1.6 Pause on Errors

The -P switch causes ASM to pause after displaying errors for a line. It will wait until the ENTER key is pressed.

SECTION 3

Assembling Programs

43

3.1.1.7 Set Symbol Table Size

The -S# switch sets the size of the symbol table. The pound sign stands for an unsigned decimal integer indicating the maximum number of symbols that the table is to hold. Since performance degrades as the symbol table approaches capacity, reserve some unused space. The default table size is 1000 symbols.

Whatever memory is left over after allocating the symbol table is used for macro buffer space. So the larger the symbol table, the less space there is for macro definitions; the smaller the symbol table, the more space there is for macros.

If you need to assemble programs with more than about 800 symbols, use -S# to increase the size of the symbol table. On the other hand, if you get a "Macro Buffer Overflow" error, try decreasing the symbol table size.

3.1.2 Examples

ASM PROG	Assemble PROG.ASM from the default drive and directory, generating PROG.OBJ on the same drive in the same directory. Do not produce a listing and do not pause on errors.
ASM \XXX\PROG -L -P	Assemble PROG.ASM from directory \XXX of the default drive, generating PROG.OBJ on the same drive in the same directory. Produce a listing on the screen and pause on errors.
ASM P1 B:P.OBJ -NM	Assemble P1.ASM from the default drive and directory, generating P.OBJ in the default directory of drive B. Do not produce a listing, do not pause on errors, and do not perform macro processing.
ASM PROG -L >LPT1:	Assemble PROG.ASM from the default drive and directory, generating PROG.OBJ on the same drive in the same directory. Produce a listing on LPT1 and do not pause on errors.

3.1.3 Assembly Status Messages

pass 1	ASM is performing pass 1 of its algorithm.
--------	--

pass 2

ASM is performing pass 2 of its
algorithm.

nnnnn lines have errors

The number of program lines with
errors is nnnnn.

SECTION 3

Assembling Programs

44

Waiting...

After reporting an error, ASM is waiting for the ENTER key.

3.1.4 Program Error Messages

- Bad Expression An improperly formed expression was found.
- Bad Parameter A macro call specifies too many parameters.
- Bad Symbol An improperly formed symbol was found.
- Can't Open <file> The named file cannot be opened.
- Close Error A file cannot be closed properly.
- CS Not Assumed for this Segment A code segment has no ASSUME CS:...
- Deleted Object File The object file has been deleted because of program errors.
- Error in Object File An error occurred while writing the object file. Most likely there is insufficient free space on disk.
- Filename Too Long: <file> A command-line file specification is too long.
- Invalid Extension: <file> A command-line file specification contains an improper extension.
- Invalid Instruction ASM does not recognize the combination of op-code and operands.
- Missing END The input file has no END directive.
- Missing ENDM The end of the input file was found while inside a macro definition.

- Need FAR PTR

The FAR PTR operator is needed in an operand expression to prevent phase errors when a FAR forward reference is being made.

SECTION 3

Assembling Programs

45

- No Source File ASM was invoked without a source file name.
- Not Addressable An END directive contains a start address which does not reference a memory location.
- Phase Error In pass 2, a label fell on a different address than it did in pass 1 or a segment's length changed.
- Procedure Error A PROC directive was found within a procedure or an ENDP directive was found outside of a procedure.
- Range Error A self-relative reference exceeded it's maximum range. This error may be accompanied by subsequent phase errors.
- Redundant Definition A symbol is defined more than once. Only the = directive is permitted to redefine symbols, and then only the ones it defined originally.
- Relocation Error One of the unary operators ~, !, or - is applied to a relocatable item.
- Segment Error One of the many segment definition or reference rules has been broken. The possibilities are (1) a memory reference has no assumed or explicit segment, (2) a GROUP directive lists a non-segment name, (3) a SEGMENT directive was found within a segment, (4) an ENDS directive was found outside of a segment, (5) an ENDS directive does not name the open segment, (6) an ASSUME directive lists a non-segment name, (7) an attempt to define data outside of a segment, (8)

an END directive was found
inside of a segment, (9) an
expression contains a segment
override prefix but does not
reference memory, (10) an
expression contains a segment
override prefix based on a name
which is not a segment name,

SECTION 3

Assembling Programs

46

(11) an expression contains a segment override prefix based on a register which is not a segment register, and (12) an expression contains a binary operator with two operands which exist in different segments, (13) a NEAR jump or call to another segment, (14) a .CASE, .186, ... directive is not located before the first segment.

- Syntax Error

A recognizable directive is not properly formed.

- Undefined Symbol

The operand field contains a reference to an undefined symbol.

- Wrong Hardware

A recognizable instruction or instruction prefix is unknown to the target CPU or NPX.

3.1.5 Assembler Failure Messages

+ Bad Register Code for: <line>

A recognized register is unknown to enreg() which must encode it into the instruction. This error should never occur, since it implies a logic error in ASM.

+ Lost Label on Pass 2: <line>

During pass 2, a label was found in the source line, but was not found in the symbol table. This error should never occur, since it implies a logic error in ASM.

+ Macro Buffer Overflow

The macro text buffer has overflowed. Consider getting more memory for macro processing by using the -S# switch to reduce the symbol table from its default of 1000 entries to a smaller number.

+ OBJ Record Conflict

An attempt was made to write a

repeatable item into a record of the object file, but the open record is of the wrong type. This error should never occur, since it implies a logic error in ASM.

SECTION 3

Assembling Programs

47

- + obuf[] overflow An attempt was made by putobj() to exceed the object record buffer obuf[]. This error should never occur, since it implies a logic error in ASM.
- + Symbol Buffer Overflow at: <line> The symbol buffer, to which the symbol table refers, has overflowed.
- + Symbol Table Overflow at: <line> The symbol table has overflowed. Consider using the -S# switch to enlarge the number of entries the symbol table can hold from its default of 1000.
- + Too Many Segments A SEGMENT directive would define more than 10 unique segments.

SECTION 3

Assembling Programs

48

3.2 LINK: The Microsoft Linker

Usage: LINK obj... , [exe] , [map] , [lib...] [switch...] [;]

obj	Object file to be linked.
exe	Executable file to be created.
map	Map file to be created.
lib	Library file to be searched and linked.
switch	Switch that controls a LINK option.
;	Command-line terminator;

3.2.1 Description

Since Small Assembler generates object files that are compatible with the Microsoft linker and since DOS comes with the Microsoft linker, there is no need for a separate linker in the Small Assembler package.

The Microsoft linker supports a number of features which are irrelevant to the Small Assembler programmer. It also supports numerous options for specifying its operating parameters. This section explains only enough about LINK to allow its use in a fashion that is consistent with the way ASM executes. For a fuller description of LINK refer to your DOS documentation or any Microsoft language manuals you may happen to have.

Command-line arguments may be given in any order. Switches must be introduced by a slash. Brackets, above, indicate optional arguments; do not enter the brackets themselves. Ellipses indicate that other instances of the preceding item are allowed.

The four types of file that may be specified (obj, exe, map, and lib) must appear in the order shown and separated by commas. If an embedded file is omitted from the list, its surrounding commas must both be present. Trailing commas may be omitted. When more than one file of a given type is specified, their names must be separated by spaces (or plus signs). Switches may appear anywhere in the command line; they do not have to be grouped at the end as shown above. Each switch must be prefixed with a slash. File specifications may contain drive, path, and extension. The DOS defaults are used if no drive or path is given. If no extension is given, a default (described below) is assumed.

3.2.1.1 Object File

Any number of object files may be named. If more than one is given, than LINK combines them all into a single executable file. The default extension for an object file is OBJ.

SECTION 3

Assembling Programs

49

3.2.1.2 Executable File

This is the output of the linker run. It contains the final program in a format that can be handled by the DOS loader. If no executable file is specified, then the output is given the name of the first (or only) object file and an extension of EXE. The default extension for an executable file is EXE.

3.2.1.3 Map File

This optional file will contain a listing of the new program's memory-map. If no map filename is given and the /MAP switch is not given, no map listing will be created. The default extension for a map file is MAP. If a map file is not specified but the /MAP switch is given, a map listing will be generated in a file with the name of the executable file and an extension of MAP.

3.2.1.4 Library File

Libraries are used to resolve external references that remain after all references between the named object files have been resolved. Each named library is searched in the order they appear in the command line. When a module containing one or more public symbols that satisfy unresolved references is found, that module is copied from the library and combined with the program. Its public symbols are used to resolve program references, and if more unresolved references remain the search continues. When every external reference has been resolved, LINK completes its work and terminates. When unresolved references still remain after searching all named libraries, LINK complains, finishes its work, and terminates.

The default extension for a library file is LIB.

3.2.1.5 Switches

Only the most prominent LINK switches are described here. See your DOS or Microsoft language documentation of a full listing. In the list which follows, the capitalized letters in each switch's name are the only ones that have to be specified; remaining letters are optional:

SWITCH	EFFECT
/HElp	Displays all possible switches.
/Exepack	Pack the EXE file. (no DEBUG)

/Map	Generates a map listing.
/NOIgnorecase	Makes LINK case sensitive.
/STack:size	Sets the stack to "size" bytes.

SECTION 3

Assembling Programs

50

3.2.1.6 Command-Line Terminator

A semicolon may be placed at the end of the command line. If it is omitted and any trailing file types have been omitted, LINK will prompt for the unspecified files.

3.2.2 Examples

LINK PROG,,,C:\SC\CLIB	Link PROG.OBJ with any necessary modules from the Small C library CLIB.LIB in directory \SC of drive C. Output an executable file named PROG.EXE.
------------------------	---

LINK X Y Z,,,C:\SC\CLIB /NOI	Link X.OBJ, Y.OBJ, and Z.OBJ with any necessary modules from the Small C library CLIB.LIB in directory \SC of drive C. Consider upper- and lower-case variants of a letter as distinct while resolving external references. Output an executable file named X.EXE.
------------------------------	--

3.2.3 Error Messages

The most frequently encountered LINK error messages are:

Ambiguous switch error: <switch>	A switch was not specified with enough letters to make it unique.
Cannot find library: <library>	LINK could not find the named library.
Cannot open ... file	LINK needs more disk space.
Out of space on ... file	LINK needs more disk space.
<name> is not a valid library	A file named as a library is not in the format of a library file.
Invalid object module	An object module is not in the correct format.
<sym>: Symbol defined more than once	A symbol is declared public in more than one module.

Too many ...

One of LINK's internal limits
has been exceeded.

Unrecognized switch error: <switch>

LINK does not recognize a
switch.

SECTION 3	Assembling Programs	51
-----------	---------------------	----

Unresolved externals	This message introduces a listing of the external reference symbols that could not be resolved. Matching public symbols were not found in any of the object files or libraries.
----------------------	---

Warning: No stack segment	No segment with the STACK combine type was found. This is only a warning since programs may set up their stacks at run time.
---------------------------	--

SECTION 4

Small Assembler Utilities

SECTION 4

Small Assembler Utilities

53

4.1 AR: The Source File Archive Utility

Usage: AR -{DPTUX} arcfile [file...]

-{DPTUX} Function switch.

arcfile Archive file specification.

file... List of individual filenames.

4.1.1 Description

Since Small Assembler has a large number of miscellaneous source files that augment its three main source files, the miscellaneous files are collected into a single archive file for distribution. AR may be used to gain access to these miscellaneous files individually.

AR collects separate text files into a single archive file. Individual files can be extracted from the archive, new ones added, old ones replaced or deleted, and a list of the archive contents can be produced.

NOTE: Although the archive is a pure ASCII file, you must not edit it because that would change the length if its members and thereby trick AR into missing the member boundaries.

The first argument must be a switch telling AR which of its functions to perform. One and only one switch is required. The second argument must be the name of an archive file. The third and subsequent arguments are filenames.

While AR is running, a control-S from the keyboard will cause it to pause execution and a control-C will cause it to abort. Press the ENTER key to resume execution after a pause.

4.1.2 The Delete Switch

The -D switch commands AR to delete one or more files from the archive. Filenames appearing after the archive name specify the archive files to be deleted.

4.1.3 The Print Switch

The -P switch commands AR to print one or more files from the archive. Filenames appearing after the archive name specify the archive files to be printed. If no names are given, all files are printed. Output goes to standard output file, and so may be redirected to any file or device.

SECTION 4

Small Assembler Utilities

54

4.1.4 The Table-of-Contents Switch

The -T switch commands AR to print the names of one or more files which the archive contains. Filenames appearing after the archive name specify the archive filenames to be listed. If no names are given, all filenames are listed. Output goes to standard output file, and so may be redirected to any file or device.

4.1.5 The Update Switch

The -U switch commands AR to update (add or replace) one or more files in the archive. Filenames appearing after the archive name specify the archive files to be updated. If no names are given, names are taken from the standard input file. They may be entered (one per line) from the keyboard or they may come from a redirection file.

For each name, AR looks for the designated file on disk and copies it into the archive file, replacing a preexisting file of the same name if necessary.

This is the command used to create a new archive file.

4.1.6 The Extract Switch

The -X switch commands AR to extract (copy) one or more files from the archive to disk. Filenames appearing after the archive name specify the archive files to be copied. If no names are given, all files are copied. The contents of the archive are not affected by this command.

4.1.7 The Archive Filename

An archive file must be named immediately after the function switch. It may contain a drive and path, but must specify the extension if the archive file has one. If an update function is being performed and there is no archive by the designated name, a new archive file will be created.

4.1.8 The Individual Filenames

Filenames listed after arcfile designate which particular files within the archive are subject to the function being performed. These filenames must include their extensions, if they have extensions.

4.2 DUMP: The File Dump Utility

Usage: DUMP [file.ext]

file name of the file to be dumped.

SECTION 4

Small Assembler Utilities

55

ext filename extension of the file to be dumped.

4.2.1 Description

DUMP produces a formatted listing of the contents of a file. OBJ and EXE files are specially formatted to make their contents intelligible. Other type files are dumped in standard hex/ASCII format. Output goes to the standard output file and, therefore, may be redirected to a printer, disk file, or whatever.

The file to be dumped must be specified, with its extension. If no file is given in the command line, DUMP prompts for one. DUMP uses the extension of the filename to determine the type of formatting to perform. The filename may include drive and path specifications.

4.2.2 Examples

4.2.2.1 DUMP XYZ.C

file: XYZ.C

```
0000 23 69 6E 63 6C 75 64 65-20 3C 73 74 64 69 6F 2E #include <stdio.
0010 68 3E 0D 0A 0D 0A 69 6E-74 20 69 3B 0D 0A 75 6E h>....int i;..un
0020 73 69 67 6E 65 64 20 63-68 61 72 20 73 74 72 5B signed char str[
0030 38 30 5D 3B 0D 0A 75 6E-73 69 67 6E 65 64 20 63 80];..unsigned c
```

4.2.2.2 DUMP XYZ.OBJ

file: XYZ.OBJ

T-MODULE HEADER RECORD: module name = XYZ.OBJ

LIST OF NAMES RECORD:

```
index = 1 name =
index = 2 name = CODE
index = 3 name = DATA
```

SEGMENT DEFINITION RECORD:

```
Name Indexes: Segment = 2 Class = 1 Overlay = 1
Segment length = 1FD
Attributes: PARA aligned (3)
```


combine as PUBLIC (2)
length < 64K

SEGMENT DEFINITION RECORD:

Name Indexes: Segment = 3 Class = 1 Overlay = 1

Segment length = 136

Attributes: PARA aligned (3)

combine as PUBLIC (2)
length < 64K

SECTION 4

Small Assembler Utilities

56

EXTERNAL NAMES DEFINITION RECORD:

```
type index =    0    name = __MAIN
type index =    0    name = _ATOP
type index =    0    name = _FT2S
```

PUBLIC NAMES DEFINITION RECORD:

```
group index =    0    segment index =    2
type index =    0    offset =    2    public Name = _I
type index =    0    offset =    4    public Name = _STR
type index =    0    offset =   5E    public Name = _FT
```

PUBLIC NAMES DEFINITION RECORD:

```
group index =    0    segment index =    1
type index =    0    offset =    2    public Name = _MAIN
type index =    0    offset =   88    public Name = _PRINT
```

```
LOGICAL ENUMERATED DATA RECORD: segment index =    1    offset =    0
0000 0000 00 00                                ..
```

```
LOGICAL ENUMERATED DATA RECORD: segment index =    2    offset =    0
0000 0000 00 00                                ..
```

```
LOGICAL ITERATED DATA RECORD: segment index =    2    offset =    2
repeat count =    1    block count =    0
count =    2
0002 0000 00 00                                ..
repeat count =   50    block count =    0
count =    1
0004 0000 00
repeat count =    A    block count =    0
count =    1
0005 0000 00
repeat count =    A    block count =    0
count =    1
0006 0000 00
repeat count =    8    block count =    0
count =    1
0007 0000 00
repeat count =    4    block count =    0
count =    1
0008 0000 00                                .
```

```
LOGICAL ENUMERATED DATA RECORD: segment index =    1    offset =    2
0002 0000 55 8B EC B8 00 00 50 B1-01 E8 00 00 83 C4 02 B8
```

U.....P.....

```
0012 0010 00 00 50 B8 50 00 50 33-C0 50 B1 03 E8 00 00 83
```

..P.P.P3.P.....

```
      0022 0020 C4 06 B8 00 00 50 B8 00-00 50 B8 00 00 50 B1 02
.....P...P...P..
      0032 0030 E8 00 00 83 C4 04 50 B1-02 E8 00 00 83 C4 04 B8
.....P.....
      0042 0040 00 00 50 B8 00 00 50 B8-00 00 50 B1 02 E8 00 00
..P...P...P.....
      0052 0050 83 C4 04 50 B1 02 E8 00-00 83 C4 04 B8 00 00 50
...P.....P
      0062 0060 B8 00 00 50 B1 02 E8 00-00 83 C4 04 B8 00 00 50
...P.....P
      0072 0070 B8 00 00 50 B1 02 E8 00-00 83 C4 04 32 C9 E8 00
...P.....2...
      0082 0080 00 EB 80 90 5D C3                      ....].
```

SECTION 4

Small Assembler Utilities

57

```

FIXUPP RECORD:      off                      fndx tndx tdis
FIXUP: M_SEG  L_OFF    4 F_SI   T_SID      2   2   74
FIXUP: M_SELF L_OFF    A F_EI   T_EI0      6   6
FIXUP: M_SEG  L_OFF   10 F_SI   T_SID      2   2   4
FIXUP: M_SELF L_OFF   1D F_EI   T_EI0      7   7
FIXUP: M_SEG  L_OFF   23 F_SI   T_SID      2   2   7B
FIXUP: M_SEG  L_OFF   27 F_SI   T_SID      2   2   5E
FIXUP: M_SEG  L_OFF   2B F_SI   T_SID      2   2   4
FIXUP: M_SELF L_OFF   31 F_EI   T_EI0      5   5

MODULE END RECORD:  off                      fndx tndx tdis
Non-Main, No Start Address

```

The preceding hex/ASCII dump is preceded by two offset fields -- the offset from the beginning of the segment and the offset of from the beginning of the data record. Since space is limited on an 80-column page, only the low order 16 bits of a 32-bit offset are shown even though an LED386 record is being displayed.

For a description of object record formats and the meanings of the symbols used in this dump, see Object Files (Section 2.2).

4.2.2.3 DUMP XYZ.EXE

```
file: XYZ.EXE
```

```

5A4D EXE signature
1B0 bytes in last page
1A 512-byte pages in file
4 relocation entries
20 paragraphs in header
0 min paragraphs after program
FFFF max paragraphs after program
317 SS before relocation
40 SP
93B2 negative chksum
2A90 IP
0 CS before relocation
1E bytes to relocation table
0 overlay number
01 00
relocation table:
0000:2A91 0000:2A9A 0000:2AB2 0000:2AB5

```

waiting...

```
0000 00 00 55 8B EC B8 74 00-50 B1 01 E8 BE 1B 83 C4 ..U...t.P.....
0010 02 B8 04 00 50 B8 50 00-50 33 C0 50 B1 03 E8 11 ....P.P.P3.P....
0020 1A 83 C4 06 B8 7B 00 50-B8 5E 00 50 B8 04 00 50 .....{.P.^..P...P
0030 B1 02 E8 6D 0A 83 C4 04-50 B1 02 E8 8E 1B 83 C4 ...m....P.....
```

SECTION 4

Small Assembler Utilities

58

If output is to the console, after displaying the EXE header, DUMP waits for the ENTER key before proceeding. This prevents the header information from scrolling off the screen before it can be studied.

4.2.3 Messages

can't find: <file>	The specified file cannot be found.
waiting...	DUMP is waiting for the ENTER key.
not a valid EXE file	The EXE file being dumped has an invalid header.
- Unknown record type: <type>	The OBJ file being dumped contains an unrecognizable record type.
- Bad record length	The length of an OBJ record does not match its length field.
- Bad checksum: %2x	The checksum of an OBJ record does not match the calculated checksum.
- Abnormal End of File	The OBJ file being dumped ends without a MODEND record.

4.3 CMIT: The Configuration Utility

Usage: CMIT [-#] [-O] [-L] [table]

-#	Highest numbered CPU to recognize.
-O	Output the machine instruction table as an OBJ file.
-L	List the compiled machine instruction table.
table	Name of the Machine Instruction Table source file.

4.3.1 Description

A machine instruction table (MIT) is used to "teach" ASM how to recognize machine instructions and how to generate object code. The standard MIT (80X86.MIT) contains the definitions of the entire family of 8086 family CPUs and NPXs. This is an ASCII text file which serves as input to CMIT the utility which "Compiles Machine Instruction Tables."

CMIT translates MITs from source to object format for loading with ASM. Section 2.3 describes the source format. Appendix A lists the standard MIT which is supplied with the Small Assembler package.

Once a table has been compiled, CMIT may list it and/or copy it into an OBJ file.

SECTION 4

Small Assembler Utilities

59

The listing produced by CMIT is produced from the object table. The source table is read a second time and each instruction is looked up in the internal MIT, using the same functions as the assembler uses. This guarantees that the table is serviceable.

Each instruction is looked up and listed with its source, the number of looks needed to find it, and the object code which will be generated when the instruction is assembled.

NOTE: Whenever a new MIT is created, its listing must be carefully checked to verify that it will generate the correct object code.

Switches may be introduced with either hyphens (-) or slashes (/).

4.3.1.1 The -# Switch

A hyphen or slash followed by a single decimal digit specifies the highest CPU and NPX in the 8086 family that CMIT will handle. Any instruction for a processor that is later in the series is ignored. The valid values of this switch are:

SWITCH	CPU	NPX
-0	8086	8087
-1	80186	8087
-2	80286	80287
-3	80386	80387

4.3.1.2 The -O Switch

The -O switch causes CMIT to generate an object file with the same name as the source file but with an OBJ extension.

4.3.1.3 The -L Switch

The -L switch causes CMIT to test the the compiled table and list it to the standard output file. The output may be redirected to a printer, disk file, or whatever.

If no switches are given, -L is assumed. However, if any switches are given, only requested actions are taken.

4.3.1.4 The Source File

If no source file is named, 80X86.MIT is assumed. A filename without an extension or with an extension of .MIT designates a different MIT source file. Any extension other than MIT will be rejected. A drive and/or path may be specified.

SECTION 4

Small Assembler Utilities

60

4.3.2 Examples

CMIT Compile 80X06.MIT from the default drive and path, and list the resulting table on the screen.

CMIT -O -L >80X86.LST Compile 80X06.MIT from the default drive and path, generate an object file 80X86.OBJ (also on the default drive and path), and list the resulting table in file 80X86.LST.

4.3.3 CMIT Status Messages

The object form of the machine instruction table contains three separate tables and one buffer. Mntbl[] contains an entry for each legal mnemonic code. Optbl[] contains an entry for each legal sequence of operands. Mitbl[] contains an entry for each legal combination of mntbl[] and optbl[] entries.

Mitbuf[] is an amalgamated buffer of strings pointed to by the three tables. Mntbl[] entries point to legal mnemonic strings in mitbuf[]. Optbl[] entries point to legal operand type strings in mitbuf[]. And mitbl[] entries point to code generation strings in mitbuf[].

The status messages tell how much of these structures was used to contain the object MIT. This information can be used to scale down the size of these structures so as to reduce wasted space. Take care not to reduce the tables too much. The search technique used on them requires evenly distributed empty space for reducing the time required to resolve synonym clashes. Probably about 20% empty space in each table is sufficient. The buffer requires no such considerations.

mntbl[]: used <u> of <s> dwords The number of double words actually used in mntbl[] is indicated by the number <u>. The size of mntbl[] is <s> double words.

optbl[]: used <u> of <s> dwords The number of double words actually used in optbl[] is indicated by the number <u>. The size of optbl[] is <s> double words.

mitbl[]: used <u> of <s> dwords The number of double words actually used in mitbl[] is indicated by the number <u>. The size of mitbl[] is <s> double words.

mitbuf[]: used <u> of <s> bytes The number of bytes actually used in
mitbuf[] is indicated by the number
<u>. The size of mitbuf[] is <s>
bytes.

SECTION 4

Small Assembler Utilities

61

4.3.4 Error Messages

- | | |
|-------------------------------------|--|
| - Bad Hardware Code in: <line> | The hardware code in <line> is not valid. |
| - Bad Hex Byte in: <line> | An improper hex byte was found in the object field of <line>. |
| - Bad Operand Field in: <line> | An invalid symbol was found in the operand field of <line>. |
| - Can't Find Mnemonic in: <line> | There is no mnemonic field in <line>. |
| - Can't Find Mnemonic in mntbl[] | The search algorithm was unable to find a mnemonic code in the compiled table. |
| - Can't Find Operand in: <line> | There is no operand field in <line>. |
| - Can't Find Operand in optbl[] | The search algorithm was unable to find an operand sequence in the compiled table. |
| - Can't Find Instruction in mitbl[] | The search algorithm was unable to find the combined mnemonic code and operand sequence in the compiled table. |
| - Field Buffer Overflow in get_mn() | The buffer that holds individual mnemonic codes which are extracted from the source line has overflowed. |
| - optbl[] Overflow on: <line> | The size of optbl[] is too small to contain all of the operand sequences. It overflowed on the indicated line. |
| - mitbl[] Overflow on: <line> | The size of mitbl[] is too small to contain all of the machine instructions (mnemonic/operand combinations). It overflowed |

	on the indicated line.
- mitbuf[] Overflow	The size of mitbuf[] is too small.

SECTION 4

Small Assembler Utilities

62

- mntbl[] Overflow on: <line>

The size of mntbl[] is too small to contain all of the mnemonics. It overflowed on the indicated line.

- Unexpected Format Byte: <byte>

An undefined byte was found in the string of code generating commands to which the found mitbl[] entry points.

63

APPENDIX A

The Machine Instruction Table

;----- MACHINE INSTRUCTIONS -----				
;first 80x87				
;				
; first 80x86				
;				
; op-codes mnemonics operands descriptions				
;----- prefixes				
00 F0	+LOCK			; Assert LOCK# signal
00 F3	+REP			; Repeat until CX=0 or ZF!=0 (note 2)
00 F3	+REPE			; Repeat until CX=0 or ZF!=0 (note 2)
00 F3	+REPZ			; Repeat until CX=0 or ZF!=0 (note 2)
00 F2	+REPNE			; Repeat until CX=0 or ZF=0 (note 2)
00 F2	+REPNZ			; Repeat until CX=0 or ZF=0 (note 2)
00 2E	+CS:			; code segment override
00 36	+SS:			; stack segment override
00 3E	+DS:			; data segment override
00 26	+ES:			; E data segment override
03 64	+FS:			; F data segment override
03 65	+GS:			; G data segment override
03 67	+ASO			; Address size override (note 1)
03 66	+OSO			; Opnd size override (note 1)
;----- CPU instructions				
00 37	AAA			; ASCII adjust AL after addition
00 D5_0A	AAD			; ASCII adjust AX before division
00 D4_0A	AAM			; ASCII adjust AX after multiply
00 3F	AAS			; ASCII adjust AL after subtraction
00 14_i2	ADC	AL,i8		; Add with CF i b to AL
00 15_i2	ADC	eAX,i1632		; Add with CF i w d to eAX
00 80_/21_i2	ADC	/8,i8		; Add with CF i b to r/m b
00 83_/21_i2	ADC	/1632,i8		; Add with CF s-ext i b to r/m w d
00 81_/21_i2	ADC	/1632,i1632		; Add with CF i w dw to r/m w
00 10_/r21	ADC	/8,r8		; Add with CF r b to r/m b
00 11_/r21	ADC	/1632,r1632		; Add with CF r w d to r/m w d
00 12_/r12	ADC	r8,/8		; Add with CF r/m b to r b
00 13_/r12	ADC	r1632,/1632		; Add with CF r/m w d to r w d
00 04_i2	ADD	AL,i8		; Add i b to AL
00 05_i2	ADD	eAX,i1632		; Add i w d to eAX
00 80_/01_i2	ADD	/8,i8		; Add i b to r/m b
00 83_/01_i2	ADD	/1632,i8		; Add s-ext i b to r/m w d
00 81_/01_i2	ADD	/1632,i1632		; Add i w d to r/m w d
00 00_/r21	ADD	/8,r8		; Add r b to r/m b

```
00 01_/r21      ADD      /1632,r1632      ; Add r w|d to r/m w|d
00 02_/r12      ADD      r8,/8            ; Add r/m b to r b
00 03_/r12      ADD      r1632,/1632      ; Add r/m w|d to r w|d
00 24_i2        AND      AL,i8            ; AND i b to AL
00 25_i2        AND      eAX,i1632        ; AND i w|d to eAX
00 80_/41_i2    AND      /8,i8            ; AND i b to r/m b
00 83_/41_i2    AND      /1632,i8         ; AND s-ext i b to r/m w|d
00 81_/41_i2    AND      /1632,i1632     ; AND i w|d to r/m w|d
00 20_/r21      AND      /8,r8            ; AND r b to r/m b
```


64

```

00 21_/r21      AND      /1632,r1632      ; AND r w|d to r/m w|d
00 22_/r12      AND      r8,/8            ; AND r/m b to r b
00 23_/r12      AND      r1632,/1632      ; AND r/m w|d to r w|d
02 63_/r21      ARPL     /16,r16          ; Adjust RPL of r/m16 to >= r16
01 62_/r12      BOUND    r1632,m3264      ; INT 5 if r1632 not in bounds
03 0F_BC_/r12   BSF      r1632,/1632      ; Bit Scan Forward on r/m w|d
03 0F_BD_/r12   BSR      r1632,/1632      ; Bit Scan Reverse on r/m w|d
03 0F_A3_/r21   BT       /1632,r1632      ; Bit Test (CF <- bit)
03 0F_BA_/41_i2 BT       /1632,i8         ; Bit Test (CF <- bit)
03 0F_BB_/r21   BTC      /1632,r1632      ; Bit Test (CF <- bit) and complement
03 0F_BA_/71_i2 BTC      /1632,i8         ; Bit Test (CF <- bit) and complement
03 0F_B3_/r21   BTR      /1632,r1632      ; Bit Test (CF <- bit) and reset
03 0F_BA_/61_i2 BTR      /1632,i8         ; Bit Test (CF <- bit) and reset
03 0F_AB_/r21   BTS      /1632,r1632      ; Bit Test (CF <- bit) and set
03 0F_BA_/51_i2 BTS      /1632,i8         ; Bit Test (CF <- bit) and set
00 E8_$1        CALL     $1632            ; Call near, self-relative
00 FF_/21        CALL     /1632           ; Call near, r/m indirect
00 9A_p1         CALL     p3248           ; Call far, direct by procedure name
00 FF_/31        CALL     m3248           ; Call far, indirect by data name
00 98            CBW          ; eAX <- s-ext AL|AX
03 98            CWDE         ; eAX <- s-ext AL|AX
00 F8            CLC          ; Clear Carry Flag
00 FC            CLD          ; Clear Direction Flag
00 FA            CLI          ; Clear Interrupt Flag
02 0F_06         CLTS        ; Clear Task-Switch Flag
00 F5            CMC          ; Complement Carry Flag
00 3C_i2         CMP         AL,i8         ; Compare i b to AL
00 3D_i2         CMP         eAX,i1632     ; Compare i w|d to eAX
00 80_/71_i2     CMP         /8,i8         ; Compare i b to r/m b
00 83_/71_i2     CMP         /1632,i8      ; Compare s-ext i b to r/m w|d
00 81_/71_i2     CMP         /1632,i1632   ; Compare i w|d to r/m w|d
00 38_/r21       CMP         /8,r8         ; Compare r b to r/m b
00 39_/r21       CMP         /1632,r1632   ; Compare r w|d to r/m w|d
00 3A_/r12       CMP         r8,/8         ; Compare r/m b to r b
00 3B_/r12       CMP         r1632,/1632   ; Compare r/m w|d to r w|d
03 A6            CMPS        m8,m8         ; Compare bytes ES:eDI with DS:eSI
01 A6            CMPSB       ; Compare bytes ES:eDI with DS:eSI
03 A7            CMPS        m1632,m1632   ; Compare words ES:eDI with DS:eSI
01 A7            CMPSW       ; Compare words ES:eDI with DS:eSI
03 A7            CMPSD       ; Compare dwords ES:eDI with DS:eSI
00 99            CWD          ; eDX:eAX <- s-ext eAX
03 99            CDQ          ; eDX:eAX <- s-ext eAX
00 27            DAA          ; Decimal adjust AL after addition
00 2F            DAS          ; Decimal adjust AL after subtraction
00 FE_/11       DEC         /8            ; Decrement r/m b by 1
00 FF_/11       DEC         /1632         ; Decrement r/m w|d by 1
00 48+r1        DEC         r1632         ; Decrement r w|d by 1

```

00 F6_/61	DIV	/8	; Div uns AX by r/m b
00 F7_/61	DIV	/1632	; Div uns eDX:eAX by r/m w d
01 C8_i1_i2	ENTER	i16,i8	; make proc parameter stack frame
00 D8_+i1_/i12	ESC	i6,/8	; Escape to other processor with x6
00 F4	HLT		; Halt
00 F6_/71	IDIV	/8	; Div sgn AX by r/m b to AL (AH=rem)
00 F7_/71	IDIV	/1632	; Div sgn eDX:eAX by r/m w d
00 F6_/51	IMUL	/8	; Mult sgn AL by r/m b to AX
00 F7_/51	IMUL	/1632	; Mult sgn eAX by r/m w d

65

```

03 0F_AF_/r12    IMUL    r1632,/1632      ; Mult sgn r w/d by r/m w/d to r w/d
01 6B_/r12_i3    IMUL    r1632,/1632,i8   ; Mult sgn r/m w/d by s-ext i b to r
w/d
01 6B_/r1_i2      IMUL    r1632,i8        ; Mult sgn r w/d by s-ext i b to r w/d
01 69_/r12_i3     IMUL    r1632,/1632,i1632 ; Mult sgn r/m w/d by i w/d to r w/d
01 69_/r1_i2      IMUL    r1632,i1632     ; Mult sgn r w/d by i w/d to r w/d
00 E4_i2          IN      AL,i8           ; Input b from i port to AL
00 E5_i2          IN      eAX,i8          ; Input w/d from i port to eAX
00 EC             IN      AL,DX           ; Input b from port DX to AL
00 ED             IN      eAX,DX          ; Input w/d from port DX to eAX
00 FE_/01         INC     /8              ; Increment r/m b by 1
00 FF_/01         INC     /16             ; Increment r/m w by 1
03 FF_/01         INC     /32             ; Increment r/m d by 1
00 40+r1          INC     r1632           ; Increment r w/d by 1
01 6C             INS     /8,DX           ; Input b from port DX into ES:DI
01 6C             INSB    ; Input b from port DX into ES:DI
01 6D             INS     /1632,DX        ; Input w/d from port DX into ES:DI
01 6D             INSW    ; Input w from port DX into ES:DI
03 6D             INSD    ; Input d from port DX into ES:EDI
00 CC             INT     3               ; Interrupt 3, trap to debugger
00 CD_i1          INT     i8              ; Interrupt numbered by i b
00 CE             INTO    ; Interrupt 4, overflow flag set
00 CF             IRET    ; Interrupt return (far & pops flags)
03 CF             IRETD   ; Interrupt return (far & pops flags)
00 77_$1          JA      $8              ; Jump short if uns >
00 73_$1          JAE     $8              ; Jump short if uns >=
00 72_$1          JB      $8              ; Jump short if uns <
00 76_$1          JBE     $8              ; Jump short if uns <=
00 72_$1          JC      $8              ; Jump short if carry
00 E3_$1          JCXZ    $8              ; Jump short if CX is 0
03 E3_$1          JECXZ   $8              ; Jump short if ECX is 0
00 74_$1          JE      $8              ; Jump short if      =
00 7F_$1          JG      $8              ; Jump short if sgn >
00 7D_$1          JGE     $8              ; Jump short if sgn >=
00 7C_$1          JL      $8              ; Jump short if sgn <
00 7E_$1          JLE     $8              ; Jump short if sgn <=
00 76_$1          JNA     $8              ; Jump short if uns <=
00 72_$1          JNAE    $8              ; Jump short if uns <
00 73_$1          JNB     $8              ; Jump short if uns >=
00 77_$1          JNBE    $8              ; Jump short if uns >
00 73_$1          JNC     $8              ; Jump short if not carry
00 75_$1          JNE     $8              ; Jump short if      !=
00 7E_$1          JNG     $8              ; Jump short if sgn <=
00 7C_$1          JNGE    $8              ; Jump short if sgn <
00 7D_$1          JNL     $8              ; Jump short if sgn >=
00 7F_$1          JNLE    $8              ; Jump short if sgn >
00 71_$1          JNO     $8              ; Jump short if not overflow

```

00 7B_\$1	JNP	\$8	; Jump short if not parity even
00 79_\$1	JNS	\$8	; Jump short if not sign
00 75_\$1	JNZ	\$8	; Jump short if not zero
00 70_\$1	JO	\$8	; Jump short if overflow
00 7A_\$1	JP	\$8	; Jump short if parity even
00 7A_\$1	JPE	\$8	; Jump short if parity even
00 7B_\$1	JPO	\$8	; Jump short if parity odd
00 78_\$1	JS	\$8	; Jump short if sign
00 74_\$1	JZ	\$8	; Jump short if zero
03 0F_87_\$1	JA	\$1632	; Jump near if uns >

66

03 0F_83_\$1	JAE	\$1632	; Jump near if uns >=
03 0F_82_\$1	JB	\$1632	; Jump near if uns <
03 0F_86_\$1	JBE	\$1632	; Jump near if uns <=
03 0F_82_\$1	JC	\$1632	; Jump near if carry
03 0F_84_\$1	JE	\$1632	; Jump near if =
03 0F_8F_\$1	JG	\$1632	; Jump near if sgn >
03 0F_8D_\$1	JGE	\$1632	; Jump near if sgn >=
03 0F_8C_\$1	JL	\$1632	; Jump near if sgn <
03 0F_8E_\$1	JLE	\$1632	; Jump near if sgn <=
03 0F_86_\$1	JNA	\$1632	; Jump near if uns <=
03 0F_82_\$1	JNAE	\$1632	; Jump near if uns <
03 0F_83_\$1	JNB	\$1632	; Jump near if uns >=
03 0F_87_\$1	JNBE	\$1632	; Jump near if uns >
03 0F_83_\$1	JNC	\$1632	; Jump near if not carry
03 0F_85_\$1	JNE	\$1632	; Jump near if !=
03 0F_8E_\$1	JNG	\$1632	; Jump near if sgn <=
03 0F_8C_\$1	JNGE	\$1632	; Jump near if sgn <
03 0F_8D_\$1	JNL	\$1632	; Jump near if sgn >=
03 0F_8F_\$1	JNLE	\$1632	; Jump near if sgn >
03 0F_81_\$1	JNO	\$1632	; Jump near if not overflow
03 0F_8B_\$1	JNP	\$1632	; Jump near if not parity even
03 0F_89_\$1	JNS	\$1632	; Jump near if not sign
03 0F_85_\$1	JNZ	\$1632	; Jump near if not zero
03 0F_80_\$1	JO	\$1632	; Jump near if overflow
03 0F_8A_\$1	JP	\$1632	; Jump near if parity even
03 0F_8A_\$1	JPE	\$1632	; Jump near if parity even
03 0F_8B_\$1	JPO	\$1632	; Jump near if parity odd
03 0F_88_\$1	JS	\$1632	; Jump near if sign
03 0F_84_\$1	JZ	\$1632	; Jump near if zero
00 EB_\$1	JMP	\$8	; Jump near short, self-rel
00 E9_\$1	JMP	\$1632	; Jump near, self-relative
00 FF_/41	JMP	/1632	; Jump near, r/m indirect
00 EA_p1	JMP	p3248	; Jump far, direct by procedure name
00 FF_/51	JMP	m3248	; Jump far, indirect data name
00 9F	LAHF		; Load FLAGS to AH
02 0F_02_/r12	LAR	r1632,/1632	; Load access rights
00 8D_/r12	LEA	r1632,m	; Load eff addr of m to r
01 C9	LEAVE		; Leave procedure
02 0F_01_/21	LGDT	m48	; Load m48 into GDTR
02 0F_01_/31	LIDT	m48	; Load m48 into IDTR
00 C5_/r12	LDS	r1632,m3248	; Load DS:r1632 with m pointer
03 0F_B2_/r12	LSS	r1632,m3248	; Load SS:r1632 with m pointer
00 C4_/r12	LES	r1632,m3248	; Load ES:r1632 with m pointer
03 0F_B4_/r12	LFS	r1632,m3248	; Load FS:r1632 with m pointer
03 0F_B5_/r12	LGS	r1632,m3248	; Load GS:r1632 with m pointer
02 0F_00_/21	LLDT	/16	; Load r/m w to LDTR
02 0F_01_/61	LMSW	/16	; Load r/m w to machine status word

00 AC	LODS	m8	; Load DS:eSI b into AL
00 AC	LODSB		; Load DS:eSI b into AL
00 AD	LODS	m1632	; Load DS:eSI w\ d into eAX
00 AD	LODSW		; Load DS:eSI w\ d into eAX
03 AD	LODSD		; Load DS:eSI w\ d into eAX
00 E2_\$1	LOOP	\$8	; decr eCX, jump if != 0
00 E1_\$1	LOOPE	\$8	; decr eCX, jump if != 0 and ZF = 1
00 E1_\$1	LOOPZ	\$8	; decr eCX, jump if != 0 and ZF = 1
00 E0_\$1	LOOPNE	\$8	; decr eCX, jump if != 0 and ZF = 0

67

```
00 E0_$1      LOOPNZ $8          ; decr eCX, jump if != 0 and ZF = 0
02 0F_03_/r12 LSL    r1632,/1632 ; Load Segment Limit to r16
02 0F_00_/31   LTR     /16        ; Load r/m w to Task Register
00 88_/r21     MOV     /8,r8       ; Move r b to r/m b
00 89_/r21     MOV     /1632,r1632 ; Move r w/d to r/m w/d
00 8A_/r12     MOV     r8,/8       ; Move r/m b to r b
00 8B_/r12     MOV     r1632,/1632 ; Move r/m w/d to r w/d
00 8C_/s21     MOV     /16,xS      ; Move xS to r/m w
00 8E_/s12     MOV     xS,/16      ; Move r/m w to xS
00 A0_o2       MOV     AL,m8       ; Move o b to AL
00 A1_o2       MOV     eAX,m1632   ; Move o w/d to eAX
00 A2_o1       MOV     m8,AL       ; Move AL to o b
00 A3_o1       MOV     m1632,eAX   ; Move eAX to o w/d
00 B0+r1_i2    MOV     r8,i8       ; Move i b to r b
00 B8+r1_i2    MOV     r1632,i1632 ; Move i w/d to r w/d
00 C6_/01_i2   MOV     /8,i8       ; Move i b to r/m b
00 C7_/01_i2   MOV     /1632,i1632 ; Move i w/d to r/m w/d
03 0F_20_/r21  MOV     r32,CRx     ; Move CRx to r32
03 0F_22_/r12  MOV     CRx,r32     ; Move r32 to CRx
03 0F_21_/r21  MOV     r32,DRx     ; Move DRx to r32
03 0F_23_/r12  MOV     DRx,r32     ; Move r32 to DRx
03 0F_24_/r21  MOV     r32,TRx     ; Move TRx to r32
03 0F_26_/r12  MOV     TRx,r32     ; Move r32 to TRx
00 A4          MOVS    m8,m8       ; Move b DS:eSI to ES:eDI
00 A4          MOVSB    ; Move b DS:eSI to ES:eDI
00 A5          MOVS    m1632,m1632 ; Move w/d DS:eSI to ES:eDI
00 A5          MOVSW    ; Move w/d DS:eSI to ES:eDI
03 A5          MOVSD    ; Move w/d DS:eSI to ES:eDI
03 0F_BE_/r12  MOVSB    r1632,/8   ; Move r/m b s-ext to r w/d
03 0F_BF_/r12  MOVSB    r32,/16    ; Move r/m w s-ext to r d
03 0F_B6_/r12  MOVZXB   r1632,/8   ; Move r/m b z-ext to r w/d
03 0F_B7_/r12  MOVZXB   r32,/16    ; Move r/m w z-ext to r d
00 F6_/41     MUL     /8          ; Mult uns AL by r/m b
00 F7_/41     MUL     /1632       ; Mult uns eAX by r/m w/d
00 F6_/31     NEG     /8          ; 2's complement r/m b
00 F7_/31     NEG     /1632       ; 2's complement r/m w/d
00 90         NOP          ; No Operation
00 F6_/21     NOT     /8          ; 1's complement r/m b
00 F7_/21     NOT     /1632       ; 1's complement r/m w/d
00 0C_i2      OR      AL,i8       ; OR i b to AL
00 0D_i2      OR      eAX,i1632   ; OR i w/d to eAX
00 80_/11_i2  OR      /8,i8       ; OR i b to r/m b
00 83_/11_i2  OR      /1632,i8    ; OR i b s-ext to r/m w/d
00 81_/11_i2  OR      /1632,i1632 ; OR i w/d to r/m w/d
00 08_/r21    OR      /8,r8       ; OR r b to r/m b
00 09_/r21    OR      /1632,r1632 ; OR r w/d to r/m w/d
00 0A_/r12    OR      r8,/8       ; OR r/m b to r b
```

00 0B_r12	OR	r1632,/1632	; OR r/m w d to r w d
00 E6_i1	OUT	i8,AL	; Output AL to port i b
00 E7_i1	OUT	i8,eAX	; Output eAX to port i b
00 EE	OUT	DX,AL	; Output AL to port DX
00 EF	OUT	DX,eAX	; Output eAX to port DX
01 6E	OUTS	DX,/8	; Output DS:eSI b to port DX
01 6E	OUTSB		; Output DS:eSI b to port DX
01 6F	OUTS	DX,/1632	; Output DS:eSI w d to port DX
01 6F	OUTSW		; Output DS:eSI w d to port DX

68

```
03 6F          OUTSD          ; Output DS:eSI w/d to port DX
00 58+r1       POP      r1632  ; Pop top of stack to r w/d
00 8F_/01      POP      m      ; Pop top of stack to m w/d
00 1F          POP      DS     ; Pop top of stack to DS
00 07          POP      ES     ; Pop top of stack to ES
00 17          POP      SS     ; Pop top of stack to SS
03 0F_A1       POP      FS     ; Pop top of stack to FS
03 0F_A9       POP      GS     ; Pop top of stack to GS
01 61          POPA         ; Pop eDI,eSI,eBP,eSP,eBX,eDX,eCX,eAX
03 61          POPAD        ; Pop eDI,eSI,eBP,eSP,eBX,eDX,eCX,eAX
00 9D          POPF         ; Pop top of stack to FLAGS
03 9D          POPFD        ; Pop top of stack to EFLAGS
00 50+r1       PUSH      r1632 ; Push r w/d
00 FF_/61      PUSH      m     ; Push m w/d
01 6A_i1       PUSH      i8    ; Push i b
01 68_i1       PUSH      i1632 ; Push i w/d
00 0E          PUSH      CS     ; Push CS
00 16          PUSH      SS     ; Push SS
00 1E          PUSH      DS     ; Push DS
00 06          PUSH      ES     ; Push ES
03 0F_A0       PUSH      FS     ; Push FS
03 0F_A8       PUSH      GS     ; Push GS
00 60          PUSHAD       ; Push eAX,eCX,eDX,eBX,eSP,eBP,eSI,eDI
03 60          PUSHAD       ; Push eAX,eCX,eDX,eBX,eSP,eBP,eSI,eDI
00 9C          PUSHF        ; Push FLAGS
03 9C          PUSHFD       ; Push EFLAGS
00 D0_/21      RCL      /8,1   ; Rotate r/m 9 bits left once
00 D2_/21      RCL      /8,CL  ; Rotate r/m 9 bits left CL times
01 C0_/21_i2   RCL      /8,i8  ; Rotate r/m 9 bits left i b times
00 D1_/21      RCL      /1632,1 ; Rotate r/m 17|33 bits left once
00 D3_/21      RCL      /1632,CL ; Rotate r/m 17|33 bits left CL times
01 C1_/21_i2   RCL      /1632,i8 ; Rotate r/m 17|33 bits left i b times
00 D0_/31      RCR      /8,1   ; Rotate r/m 9 bits right once
00 D2_/31      RCR      /8,CL  ; Rotate r/m 9 bits right CL times
01 C0_/31_i2   RCR      /8,i8  ; Rotate r/m 9 bits right i b times
00 D1_/31      RCR      /1632,1 ; Rotate r/m 17|33 bits right once
00 D3_/31      RCR      /1632,CL ; Rotate r/m 17|33 bits right CL times
01 C1_/31_i2   RCR      /1632,i8 ; Rotate r/m 17|33 bits right i b times
00 D0_/01      ROL      /8,1   ; Rotate r/m 8 bits left once
00 D2_/01      ROL      /8,CL  ; Rotate r/m 8 bits left CL times
01 C0_/01_i2   ROL      /8,i8  ; Rotate r/m 8 bits left i b times
00 D1_/01      ROL      /1632,1 ; Rotate r/m 16|32 bits left once
00 D3_/01      ROL      /1632,CL ; Rotate r/m 16|32 bits left CL times
01 C1_/01_i2   ROL      /1632,i8 ; Rotate r/m 16|32 bits left i b times
00 D0_/11      ROR      /8,1   ; Rotate r/m 8 bits right once
00 D2_/11      ROR      /8,CL  ; Rotate r/m 8 bits right CL times
01 C0_/11_i2   ROR      /8,i8  ; Rotate r/m 8 bits right i b times
```

00 D1_/11	ROR	/1632,1	; Rotate r/m 16 32 bits right once
00 D3_/11	ROR	/1632,CL	; Rotate r/m 16 32 bits right CL times
01 C1_/11_i2	ROR	/1632,i8	; Rotate r/m 16 32 bits right i b times
00 C3+f	RET		; Return near or far
00 C3	RETN		; Return near
00 CB	RETF		; Return far
00 C2+f_i1	RET	i16	; Pop i bytes and Return near or far
00 C2_i1	RETN	i16	; Pop i bytes and Return near
00 CA_i1	RETF	i16	; Pop i bytes and Return far

69

```

00 9E          SAHF          ; Store AH to FLAGS
00 D0_/41     SAL          /8,1      ; Shift arith r/m b left once
00 D2_/41     SAL          /8,CL     ; Shift arith r/m b left CL times
01 C0_/41_i2  SAL          /8,i8     ; Shift arith r/m b left i b times
00 D1_/41     SAL          /1632,1   ; Shift arith r/m w/d left once
00 D3_/41     SAL          /1632,CL  ; Shift arith r/m w/d left CL times
01 C1_/41_i2  SAL          /1632,i8  ; Shift arith r/m w/d left i b times
00 D0_/71     SAR          /8,1      ; Shift arith r/m b right once
00 D2_/71     SAR          /8,CL     ; Shift arith r/m b right CL times
01 C0_/71_i2  SAR          /8,i8     ; Shift arith r/m b right i b times
00 D1_/71     SAR          /1632,1   ; Shift arith r/m w/d right once
00 D3_/71     SAR          /1632,CL  ; Shift arith r/m w/d right CL times
01 C1_/71_i2  SAR          /1632,i8  ; Shift arith r/m w/d right i b times
00 D0_/41     SHL          /8,1      ; Shift arith r/m b left once
00 D2_/41     SHL          /8,CL     ; Shift arith r/m b left CL times
01 C0_/41_i2  SHL          /8,i8     ; Shift arith r/m b left i b times
00 D1_/41     SHL          /1632,1   ; Shift arith r/m w/d left once
00 D3_/41     SHL          /1632,CL  ; Shift arith r/m w/d left CL times
01 C1_/41_i2  SHL          /1632,i8  ; Shift arith r/m w/d left i b times
00 D0_/51     SHR          /8,1      ; Shift arith r/m b right once
00 D2_/51     SHR          /8,CL     ; Shift arith r/m b right CL times
01 C0_/51_i2  SHR          /8,i8     ; Shift arith r/m b right i b times
00 D1_/51     SHR          /1632,1   ; Shift arith r/m w/d right once
00 D3_/51     SHR          /1632,CL  ; Shift arith r/m w/d right CL times
01 C1_/51_i2  SHR          /1632,i8  ; Shift arith r/m w/d right i b times
00 1C_i2      SBB          AL,i8     ; Sub with CF i b from AL
00 1D_i2      SBB          eAX,i1632 ; Sub with CF i w/d from eAX
00 80_/31_i2  SBB          /8,i8     ; Sub with CF i b from r/m b
00 83_/31_i2  SBB          /1632,i8  ; Sub with CF s-ext i b from r/m w/d
00 81_/31_i2  SBB          /1632,i1632 ; Sub with CF i w/d from r/m w/d
00 18_/r21    SBB          /8,r8     ; Sub with CF r b from r/m b
00 19_/r21    SBB          /1632,r1632 ; Sub with CF r w/d from r/m w/d
00 1A_/r12    SBB          r8,/8     ; Sub with CF r/m b from r b
00 1B_/r12    SBB          r1632,/1632 ; Sub with CF r/m w/d from r w/d
00 AE         SCAS         m8        ; Compare AL-ES:eDI, inc|dec eDI
00 AE         SCASB        ; Compare AL-ES:eDI, inc|dec eDI
00 AF         SCAS         m1632     ; Compare eAX-ES:eDI, inc|dec eDI
00 AF         SCASW        ; Compare eAX-ES:eDI, inc|dec eDI
03 AF         SCASD        ; Compare eAX-ES:eDI, inc|dec eDI
03 0F_97_/01  SETA         /8        ; r/m b <- 1 if uns > , else 0
03 0F_93_/01  SETAE        /8        ; r/m b <- 1 if uns >=, else 0
03 0F_92_/01  SETB         /8        ; r/m b <- 1 if uns < , else 0
03 0F_96_/01  SETBE        /8        ; r/m b <- 1 if uns <=, else 0
03 0F_92_/01  SETC         /8        ; r/m b <- 1 if carry, else 0
03 0F_94_/01  SETE         /8        ; r/m b <- 1 if      = , else 0
03 0F_9F_/01  SETG         /8        ; r/m b <- 1 if sgn > , else 0
03 0F_9D_/01  SETGE        /8        ; r/m b <- 1 if sgn >=, else 0

```

03 0F_9C_/01	SETL	/8	; r/m b <- 1 if sgn < , else 0
03 0F_9E_/01	SETLE	/8	; r/m b <- 1 if sgn <=, else 0
03 0F_96_/01	SETNA	/8	; r/m b <- 1 if uns <=, else 0
03 0F_92_/01	SETNAE	/8	; r/m b <- 1 if uns < , else 0
03 0F_93_/01	SETNB	/8	; r/m b <- 1 if uns >=, else 0
03 0F_97_/01	SETNBE	/8	; r/m b <- 1 if uns > , else 0
03 0F_93_/01	SETNC	/8	; r/m b <- 1 if not carry, else 0
03 0F_95_/01	SETNE	/8	; r/m b <- 1 if !=, else 0
03 0F_9E_/01	SETNG	/8	; r/m b <- 1 if sgn <=, else 0

70

```

03 0F_9C_/01      SETNGE /8      ; r/m b <- 1 if sgn < , else 0
03 0F_9D_/01      SETNL  /8      ; r/m b <- 1 if sgn >=, else 0
03 0F_9F_/01      SETNLE /8      ; r/m b <- 1 if sgn > , else 0
03 0F_91_/01      SETNO  /8      ; r/m b <- 1 if not overflow, else 0
03 0F_9B_/01      SETNP  /8      ; r/m b <- 1 if not parity even, else 0
03 0F_99_/01      SETNS  /8      ; r/m b <- 1 if not sign, else 0
03 0F_95_/01      SETNZ  /8      ; r/m b <- 1 if not zero, else 0
03 0F_90_/01      SETO   /8      ; r/m b <- 1 if overflow, else 0
03 0F_9A_/01      SETP   /8      ; r/m b <- 1 if parity even, else 0
03 0F_9A_/01      SETPE  /8      ; r/m b <- 1 if parity even, else 0
03 0F_9B_/01      SETPO  /8      ; r/m b <- 1 if parity odd, else 0
03 0F_98_/01      SETS   /8      ; r/m b <- 1 if sign, else 0
03 0F_94_/01      SETZ   /8      ; r/m b <- 1 if zero, else 0
02 0F_01_/01      SGDT   m       ; Store GDTR to m
02 0F_01_/11      SIDT   m       ; Store IDTR to m
03 0F_A4_/r21_i3  SHLD   /1632,r1632,i8 ; << arith r/m:r i b times into r/m
03 0F_A5_/r21     SHLD   /1632,r1632,CL ; << arith r/m:r CL times into r/m
03 0F_AC_/r21_i3  SHRD   /1632,r1632,i8 ; >> arith r:r/m i b times into r/m
03 0F_AD_/r21     SHRD   /1632,r1632,CL ; >> arith r:r/m CL times into r/m
02 0F_00_/01      SLDT   /16      ; Store LDTR to r/m w
02 0F_01_/41      SMSW   /16      ; Store MSW to r/m w
00 F9             STC      ; Set CF
00 FD             STD      ; Set DF
00 FB             STI      ; Set IF
00 AA             STOS     m8      ; Store AL in ES:eDI b, inc|dec eDI
00 AA             STOSB    ; Store AL in ES:eDI b, inc|dec eDI
00 AB             STOS     m1632   ; Store eAX in ES:eDI w|d, inc|dec eDI
00 AB             STOSW    ; Store eAX in ES:eDI w|d, inc|dec eDI
03 AB             STOSD    ; Store eAX in ES:eDI w|d, inc|dec eDI
02 0F_00_/11      STR     /16      ; Store task register to r/m w
00 2C_i2          SUB     AL,i8     ; Subtract i b from AL
00 2D_i2          SUB     eAX,i1632 ; Subtract i w|d from eAX
00 80_/51_i2      SUB     /8,i8     ; Subtract i b from r/m b
00 83_/51_i2      SUB     /1632,i8  ; Subtract s-ext i b from r/m w|d
00 81_/51_i2      SUB     /1632,i1632 ; Subtract i w|d from r/m w|d
00 28_/r21        SUB     /8,r8     ; Subtract r b from r/m b
00 29_/r21        SUB     /1632,r1632 ; Subtract r w|d from r/m w|d
00 2A_/r12        SUB     r8,/8     ; Subtract r/m b from r b
00 2B_/r12        SUB     r1632,/1632 ; Subtract r/m w|d from r w|d
00 A8_i2          TEST    AL,i8     ; Test AND i b with AL
00 A9_i2          TEST    eAX,i1632 ; Test AND i w|d with eAX
00 F6_/01_i2      TEST    /8,i8     ; Test AND i b with r/m b
00 F7_/01_i2      TEST    /1632,i1632 ; Test AND i w|d with r/m w|d
00 84_/r21        TEST    /8,r8     ; Test AND r b with r/m b
00 85_/r21        TEST    /1632,r1632 ; Test AND r w|d with r/m w|d
02 0F_00_/41      VERR    /16      ; ZF=1 if r/m segment selector readable
02 0F_00_/51      VERW    /16      ; ZF=1 if r/m segment selector writable

```

00 9B	WAIT		; Wait until BUSY pin inactive (high)
00 90+r2	XCHG	eAX,r1632	; Exchange r w\ d with eAX
00 90+r1	XCHG	r1632,eAX	; Exchange r w\ d with eAX
00 86_/r21	XCHG	/8,r8	; Exchange r b with r/m b
00 86_/r12	XCHG	r8,/8	; Exchange r b with r/m b
00 87_/r21	XCHG	/1632,r1632	; Exchange r w\ d with r/m w\ d
00 87_/r12	XCHG	r1632,/1632	; Exchange r w\ d with r/m w\ d
00 D7	XLAT	m8	; Set AL to b at DS:eBX + uns AL
00 D7	XLATB		; Set AL to b at DS:eBX + uns AL

71

```

00 34_i2      XOR      AL,i8          ; Exclusive OR i b to AL
00 35_i2      XOR      eAX,i1632     ; Exclusive OR i w|d to eAX
00 80_/61_i2  XOR      /8,i8         ; Exclusive OR i b to r/m b
00 83_/61_i2  XOR      /1632,i8      ; Exclusive OR i b s-ext to r/m w|d
00 81_/61_i2  XOR      /1632,i1632   ; Exclusive OR i w|d to r/m w|d
00 30_/r21    XOR      /8,r8         ; Exclusive OR r b to r/m b
00 31_/r21    XOR      /1632,r1632   ; Exclusive OR r w|d to r/m w|d
00 32_/r12    XOR      r8,/8         ; Exclusive OR r/m b to r b
00 33_/r12    XOR      r1632,/1632   ; Exclusive OR r/m w|d to r w|d
;----- NPX instructions
00 w6_D9_F0    F2XM1                 ; calculate 2**x
00 w6_D9_E1    FABS                   ; absolute value of ST(0)
00 w6_DE_C1    FADD                    ; add real
00 w6_D8_/01   FADD      m32          ; add short real from memory
00 w6_DC_/01   FADD      m64          ; add long real from memory
00 w6_D8_C0+r2 FADD      ST,STx       ; Add real from stack
00 w6_DC_C0+r1 FADD      STx,ST       ; Add real to stack
00 w6_DE_C0+r1 FADDP      STx,ST      ; Add real and pop stack
00 w6_DF_/41   FBLD      m80          ; load packed to stack
00 w6_DF_/61   FBSTP      m80         ; store packed and pop
00 w6_D9_E0    FCHS                   ; change sign of ST(0)
00 9B_DB_E2    FCLEX                  ; clear exceptions after WAIT
00 w6_D8_D1    FCOM                    ; compare real
00 w6_D8_/21   FCOM      m32          ; compare short real
00 w6_DC_/21   FCOM      m64          ; compare long real
00 w6_D8_D0    FCOM      ST           ; compare real with ST(0)
00 w6_D8_D0+r1 FCOM      STx          ; compare real with stack
00 w6_D8_D9    FCOMP                  ; compare real and pop
00 w6_D8_/31   FCOMP      m32          ; compare short real and pop
00 w6_DC_/31   FCOMP      m64          ; compare long real and pop
00 w6_D8_D8    FCOMP      ST           ; compare real with ST(0) and pop
00 w6_D8_D8+r1 FCOMP      STx          ; compare real with stack and pop
00 w6_DE_D9    FCOMPP                 ; compare real and pop twice
30 w6_D9_FF    FCOS                   ; cosine
00 9B_D9_F6    FDECSTP                ; decrement stack pointer
00 w6_DB_E1    FDISI                  ; disable interrupts after WAIT
00 w6_DE_F9    FDIV                    ; divide real
00 w6_D8_/61   FDIV      m32          ; divide short real memory
00 w6_DC_/61   FDIV      m64          ; divide long real memory
00 w6_D8_F0+r2 FDIV      ST,STx       ; divide real from stack
00 w6_DC_F8+r1 FDIV      STx,ST       ; divide real to stack
00 w6_DE_F8+r1 FDIVP      STx,ST      ; divide real and pop to stack
00 w6_DE_F1    FDIVR                  ; divide real reversed
00 w6_D8_/71   FDIVR      m32          ; divide short real reversed from m
00 w6_DC_/71   FDIVR      m64          ; divide long real reversed from m
00 w6_D8_F8+r2 FDIVR      ST,STx       ; divide real reversed from stack
00 w6_DC_F0+r1 FDIVR      STx,ST      ; divide real reversed to stack

```

00	w6_DE_F0+r1	FDIVRP	STx,ST	; divide real reversed and pop
00	w6_DB_E0	FENI		; enable interrupts after WAIT
00	9B_DD_C0	FFREE	ST	; free ST(0) element
00	9B_DD_C0+r1	FFREE	STx	; free x-th stack element
00	w6_DE_/01	FIADD	m16	; add word integer
00	w6_DA_/01	FIADD	m32	; add short integer
00	w6_DE_/21	FICOM	m16	; compare word integer
00	w6_DA_/21	FICOM	m32	; compare short integer
00	w6_DE_/31	FICOMP	m16	; compare word integer and pop

72

00	w6_DA_/31	FICOMP	m32	; compare short integer and pop
00	w6_DE_/61	FIDIV	m16	; divide word integer
00	w6_DA_/61	FIDIV	m32	; divide short integer
00	w6_DE_/71	FIDIVR	m16	; divide word integer reversed
00	w6_DA_/71	FIDIVR	m32	; divide short integer reversed
00	w6_DF_/01	FILD	m16	; load word integer
00	w6_DB_/01	FILD	m32	; load short integer
00	w6_DF_/51	FILD	m64	; load long integer
00	w6_DE_/11	FIMUL	m16	; multiply word integer
00	w6_DA_/11	FIMUL	m32	; multiply short integer
00	9B_D9_F7	FINCSTP		; increment stack pointer
00	9B_DB_E3	FINIT		; initialize processor after WAIT
00	w6_DF_/21	FIST	m16	; store word integer
00	w6_DB_/21	FIST	m32	; store short integer
00	w6_DF_/31	FISTP	m16	; store word integer and pop
00	w6_DB_/31	FISTP	m32	; store short integer and pop
00	w6_DF_/71	FISTP	m64	; store long integer and pop
00	w6_DE_/41	FISUB	m16	; subtract word integer
00	w6_DA_/41	FISUB	m32	; subtract short integer
00	w6_DE_/51	FISUBR	m16	; subtract word integer reversed
00	w6_DA_/51	FISUBR	m32	; subtract short integer reversed
00	w6_D9_/01	FLD	m32	; load short real to ST(0)
00	w6_DD_/01	FLD	m64	; load long real to ST(0)
00	w6_DB_/51	FLD	m80	; load temp real to ST(0)
00	w6_D9_C0+r1	FLD	STx	; load long real to ST(0)
00	w6_D9_E8	FLD1		; loac +1.0 to ST(0)
00	9B_D9_/51	FLDCW	m	; load control word
00	9B_D9_/41	FLDENV	m	; load 80x87 environment
00	w6_D9_EA	FLDL2E		; load log_base_2(e) to ST(0)
00	w6_D9_E9	FLDL2T		; load log_base_2(10) to ST(0)
00	w6_D9_EC	FLDLG2		; load log_base_10(2) to ST(0)
00	w6_D9_ED	FLDLN2		; load log_base_e(2) to ST(0)
00	w6_D9_EB	FLDPI		; load pi to ST(0)
00	w6_D9_EE	FLDZ		; load +0.0 to ST(0)
00	w6_DE_C9	FMUL		; multiply real
00	w6_D8_/11	FMUL	m32	; multiply short real from m
00	w6_DC_/11	FMUL	m64	; multiply long real from m
00	w6_D8_C8+r2	FMUL	ST,STx	; multiply real from stack
00	w6_DC_C8+r1	FMUL	STx,ST	; multiply real to stack
00	w6_DE_C8+r1	FMULP	STx,ST	; multiply real and pop to stack
00	DB_E2	FNCLEX		; clear exceptions with no WAIT
00	DB_E1	FNDISI		; disable interrupts with no WAIT
00	DB_E0	FNENI		; enable interrupts with no WAIT
00	DB_E3	FNINIT		; initialize 80x87 with no WAIT
00	9B_D9_D0	FNOP		; no operation
00	DD_/61	FNSAVE	m	; save 80x87 state with no WAIT
00	D9_/71	FNSTCW	m	; store control word with no WAIT

00	D9_/61	FNSTENV m	; store 80x87 environment with no WAIT
00	DD_/71	FNSTSW m	; store 80x87 status word with no WAIT
20	DF_E0	FNSTSW AX	; store status word in AX with no WAIT
00	w6_D9_F3	FPATAN	; partial arctangent
00	w6_D9_F8	FPREM	; partial remainder
30	w6_D9_F5	FPREM1	; partial remainder
00	w6_D9_F2	FPTAN	; partial tangent
00	w6_D9_FC	FRNDINT	; round to integer
00	9B_DD_/41	FRSTOR m	; restore 80x87 state

73

```

00 9B_DD_/61      FSAVE    m          ; save 80x87 state after WAIT
00 w6_D9_FD       FSCALE   m          ; scale
20 9B_DB_E4       FSETPM   m          ; set protected mode (.278 only)
30 w6_D9_FE       FSIN     m          ; sine
30 w6_D9_FB       FSINCOS  m          ; sine and cosine
00 w6_D9_FA       FSQRT    m          ; square root
00 w6_D9_/21      FST       m32       ; store short real
00 w6_DD_/21      FST       m64       ; store long real
00 w6_DD_D0       FST       ST        ; store real from ST(0) to ST(0)
00 w6_DD_D0+r1    FST       STx       ; store real in stack
00 9B_D9_/71      FSTCW     m          ; store control word with WAIT
00 9B_D9_/61      FSTENV   m          ; store 80x87 environment after WAIT
00 w6_D9_/31      FSTP      m32       ; store short real with WAIT
00 w6_DD_/31      FSTP      m64       ; store long real with WAIT
00 w6_DB_/71      FSTP      m80       ; store temp real with WAIT
00 w6_DD_D8+r1    FSTP      STx       ; store real in stack
00 9B_DD_/71      FSTSW     m          ; store 80x87 status word after WAIT
00 w6_DE_E9       FSUB      m          ; subtract real
00 w6_D8_/41      FSUB      m32       ; subtract short real from memory
00 w6_DC_/41      FSUB      m64       ; subtract long real from memory
00 w6_D8_E0+r2    FSUB      ST,STx    ; subtract real from stack
00 w6_DC_E8+r1    FSUB      STx,ST    ; subtract real to stack
00 w6_DE_E8+r1    FSUBP     STx,ST    ; subtract real and pop to stack
00 w6_DE_E1       FSUBR     m          ; subtract real reversed
00 w6_D8_/51      FSUBR     m32       ; subtract short real reversed from m
00 w6_DC_/51      FSUBR     m64       ; subtract long real reversed from m
00 w6_D8_E8+r2    FSUBR     ST,STx    ; subtract real reversed from stack
00 w6_DC_E0+r1    FSUBR     STx,ST    ; subtract real reversed to stack
00 w6_DE_E0+r1    FSUBRP    STx,ST    ; subtract real reversed and pop
20 9B_DF_E0       FSTSW     AX        ; store status word in AX with WAIT
00 w6_D9_E4       FTST      ST(0)    ; test ST(0)
30 w6_DD_E0+r1    FUCOM     STx       ; compare unordered
30 w6_DD_E8+r1    FUCOMP    STx       ; compare unordered and pop
30 w6_DA_E9       FUCOMPP   m          ; compare unordered and pop twice
00 9B             FWAIT     m          ; wait for last 80x87 oper to finish
00 w6_D9_E5       FXAM      ST(0)    ; examine ST(0)
00 w6_D9_C9       FXCH      ST(1)    ; exchange ST(0) with ST(1)
00 w6_D9_C8       FXCH      ST(0)    ; exchange ST(0) with ST(0)
00 w6_D9_C8+r1    FXCH      STx      ; exchange ST(0) and x-th element
00 w6_D9_F4       FXTRACT   m          ; extract exponent and significand
00 w6_D9_F1       FYL2X     m          ; calculate Y*log_base_2(x)
00 w6_D9_F9       FYL2XP1   m          ; calculate Y*log_base_2(x+1)
;-----
;
; note 1: The mnemonic value of these prefixes is tentative.
;
; note 2: This prefix should be applied only to the string instructions.

```

; The ZF condition applies only to the SCAS and CMPS instructions;
; the CPU ignores the zero flag otherwise.
;

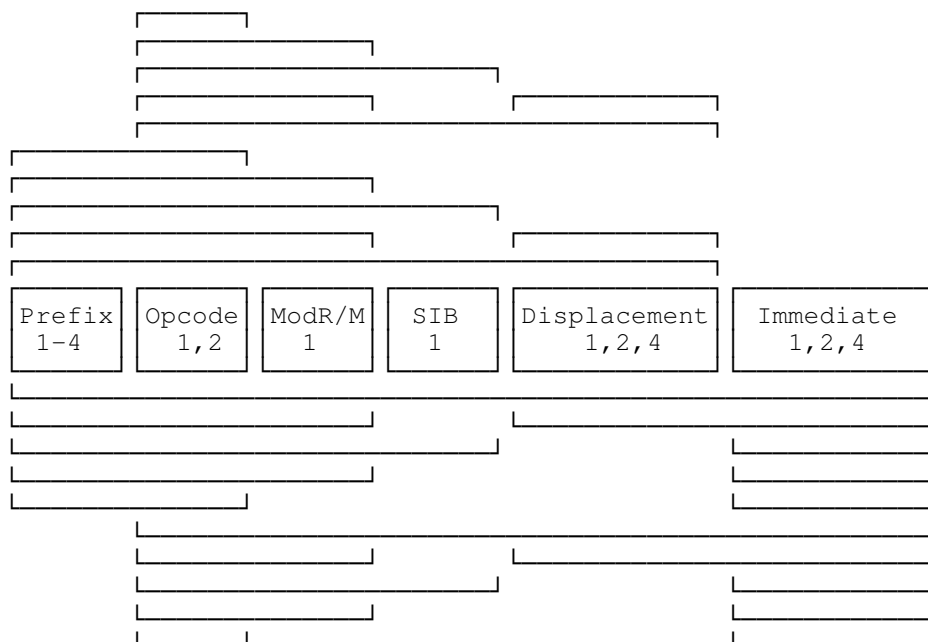
In order to fit meaningful comments in the the limited space
available, extreme abbreviation was used on certain commonly
occurring words. These are:

ABBR	MEANING
b	byte
d	double word
eXX	register XX or EXX
eDI	DI or EDI
eSI	SI or ESI
ext	extended
i	immediate
m	memory
q	quad word
r	register
s	sign
sgn	signed
uns	unsigned
w	word
z	zero
	or

APPENDIX B

Machine Instruction Formats

Instructions for the 8086 family CPUs and NPXs have a variable length format as indicated below:



The individual fields of the instruction are enclosed within boxes. Each box gives the name of the field and its possible sizes in bytes. The brackets above and below the fields are intended to show the possible combinations of fields that may constitute a legal instruction. Those below reflect the ones above except that they include the immediate operand field.

The "Prefix" field consists of zero or more prefix bytes which serve to alter the normal operation of the instruction in some way. A maximum of four prefix bytes may precede an instruction. They fall into three classes.

The "instruction" prefixes are:

PREFIX	CODE	EFFECT
F0	LOCK	Asserts the LOCK# signal during instruction execution.
F2	REPNE, REPNZ	Repeats instruction until CX=0 or (if SCAS or CMPS instruction) ZF=1.
F3	REP, REPE, REPZ	Repeats instruction until CX=0 or (if SCAS or CMPS instruction) ZF=0.

These codes are written before the instruction opcode, on the same line.

The "Segment Override" prefixes are:

PREFIX	CODE	EFFECT
2E	CS:	Calculate effective address from CS
36	SS:	Calculate effective address from SS
3E	DS:	Calculate effective address from DS
26	ES:	Calculate effective address from ES
64	FS:	Calculate effective address from FS
65	GS:	Calculate effective address from GS

These codes are written at the beginning of an operand expression.

The "Size Attribute" prefixes are:

PREFIX	CODE	EFFECT
67	ASO	Overrides the address size attribute
66	OSO	Overrides the operand size attribute

These codes are recognized by Small Assembler before the instruction opcode, on the same line. Neither the mnemonic code value or its location have been established by Microsoft since they do not yet support large segments. And, since Small Assembler object files that contain USE32 segments are not compatible with LINK, interest in these prefixes at this time is purely academic. When an operating system comes along which supports protected mode 80386 operation, Small Assembler should be ready for it with relatively little modification.

The "Opcode" field is one or two bytes of binary code which uniquely identifies the instruction.

The "ModR/M" field is a bit of a hodgepodge. The high-order two bits (the Mode bits) in combination with the low-order three bits (the Register/Memory bits) designate the method used to locate an operand. The middle three bits function as an extension of the Opcode field.

The "SIB" field was added as an extension to the ModR/M byte to support additional addressing modes when the 80386 is executing with the 32-bit address size attribute. The first two bits (the Shift bits, indicates by "ss" below) specify the number of positions by which the index value is to be shifted left (each shift multiplies by 2). The next three bits (the Index bits) identify a register from which the index value is taken. And the final three bits (the Base bits) specify a base register. The effective address is calculated as the value of the base register plus the shifted value of the index register.

When the operand is in memory, the values of the ModR/M and SIB bytes are defined as follows:

ModR/M		SIB		Operand is in Memory as Specified	
mm	... R/M	ss	iii bbb	16-bit ASA	32-bit ASA
00	... 000			DS:[BX+SI]	DS:[EAX]
00	... 001			DS:[BX+DI]	DS:[ECX]
00	... 010			SS:[BP+SI]	DS:[EDX]
00	... 011			SS:[BP+DI]	DS:[EBX]
00	... 100			DS:[SI]	
00	... 100	ss	iii 000		DS:[EAX+(i<<s)]
00	... 100	ss	iii 001		DS:[ECX+(i<<s)]
00	... 100	ss	iii 010		DS:[EDX+(i<<s)]
00	... 100	ss	iii 011		DS:[EBX+(i<<s)]
00	... 100	ss	iii 100		SS:[ESP+(i<<s)]
00	... 100	ss	iii 101		DS:[dis32+(i<<s)]
00	... 100	ss	iii 110		DS:[ESI+(i<<s)]
00	... 100	ss	iii 111		DS:[EDI+(i<<s)]
00	... 101			DS:[DI]	DS:[dis32]
00	... 110			DS:[dis16]	DS:[ESI]
00	... 111			DS:[BX]	DS:[EDI]

01	... 000			DS:[BX+SI+dis8]	DS:[EAX+dis8]
01	... 001			DS:[BX+DI+dis8]	DS:[ECX+dis8]
01	... 010			SS:[BP+SI+dis8]	DS:[EDX+dis8]
01	... 011			SS:[BP+DI+dis8]	DS:[EBX+dis8]
01	... 100			DS:[SI+dis8]	
01	... 100	ss	iii 000		DS:[EAX+(i<<s)+dis8]
01	... 100	ss	iii 001		DS:[ECX+(i<<s)+dis8]
01	... 100	ss	iii 010		DS:[EDX+(i<<s)+dis8]
01	... 100	ss	iii 011		DS:[EBX+(i<<s)+dis8]
01	... 100	ss	iii 100		SS:[ESP+(i<<s)+dis8]
01	... 100	ss	iii 101		DS:[EBP+(i<<s)+dis8]

01	...	100	ss	iii	110		DS:[ESI+(i<<s)+dis8]
01	...	100	ss	iii	111		DS:[EDI+(i<<s)+dis8]
01	...	101				DS:[DI+dis8]	SS:[EBP+dis8]
01	...	110				SS:[BP+dis8]	DS:[ESI+dis8]
01	...	111				DS:[BX+dis8]	DS:[EDI+dis8]

10	...	000				DS:[BX+SI+dis16]	DS:[EAX+dis32]
10	...	001				DS:[BX+DI+dis16]	DS:[ECX+dis32]

10 ... 010	SS:[BP+SI+dis16]	DS:[EDX+dis32]
10 ... 011	SS:[BP+DI+dis16]	DS:[EBX+dis32]
10 ... 100	DS:[SI+dis16]	
10 ... 100 ss iii 000		DS:[EAX+(i<<s)+dis32]
10 ... 100 ss iii 001		DS:[ECX+(i<<s)+dis32]
10 ... 100 ss iii 010		DS:[EDX+(i<<s)+dis32]
10 ... 100 ss iii 011		DS:[EBX+(i<<s)+dis32]
10 ... 100 ss iii 100		SS:[ESP+(i<<s)+dis32]
10 ... 100 ss iii 101		DS:[EBP+(i<<s)+dis32]
10 ... 100 ss iii 110		DS:[ESI+(i<<s)+dis32]
10 ... 100 ss iii 111		DS:[EDI+(i<<s)+dis32]
10 ... 101	DS:[DI+dis16]	SS:[EBP+dis32]
10 ... 110	SS:[BP+dis16]	DS:[ESI+dis32]
10 ... 111	DS:[BX+dis16]	DS:[EDI+dis32]

Square brackets above indicate the contents of the memory location calculated as indicated between the brackets. A register name implies its contents. The symbols "dis8," "dis16," and "dis32" above refer to the value of an 8-bit, 16-bit, or 32-bit displacement field associated with the instruction. The symbol "s" refers to the value of the Shift bits "ss" above. The symbol "i" refers to the value of the Index bits "iii" above.

The index register bits are encoded as follows:

32-Bit Index Register Codes	
iii	index register
000	EAX
001	ECX
010	EDX
011	EBX
100	none (ss must be 00)
101	EBP
110	ESI
111	EDI

When the operand is in a register, the values of the ModR/M byte is defined as follows:

Operand is in the Register Indicated					
mm ... R/M	16-bit OSA		32-bit OSA		
	byte	word	byte	word	
11 ... 000	AL	AX	AL	EAX	
11 ... 001	CL	CX	CL	ECX	
11 ... 010	DL	DX	DL	EDX	
11 ... 011	BL	BX	BL	EBX	
11 ... 100	AH	SP	AH	ESP	
11 ... 101	CH	BP	CH	EBP	
11 ... 110	DH	SI	DH	ESI	
11 ... 111	BH	DI	BH	EDI	

In the ModR/M encodings above, the middle three bits were unspecified. That was because they either served to qualify the opcode or to specify a second operand which may reside in a register. In the latter case, these bits are encoded as follows:

Second Operand is in the Register Indicated					
.. rrr ...	16-bit OSA		32-bit OSA		
	byte	word	byte	word	
.. 000 ...	AL	AX	AL	EAX	
.. 001 ...	CL	CX	CL	ECX	
.. 010 ...	DL	DX	DL	EDX	
.. 011 ...	BL	BX	BL	EBX	
.. 100 ...	AH	SP	AH	ESP	
.. 101 ...	CH	BP	CH	EBP	
.. 110 ...	DH	SI	DH	ESI	
.. 111 ...	BH	DI	BH	EDI	

As we have seen, when an instruction has two operands, one may be in memory or in a register (the one specified by the Mode and Register/Memory bits if the ModR/M byte, possibly augmented with the SIB byte) and the other must be in a register (the one specified by the middle bits of the ModR/M byte). A bit in the Opcode field specifies which of these operands is the "destination" operand (left-most in the source line) and which is the "source" operand (right-most in the source line).

The "Displacement" field may be one, two, or four bytes long. An 8-bit displacement is always sign extended before it figures into an address calculation. An instruction which ordinarily calls for a 16-bit displacement will be assembled with a 32-bit displacement if the address size attribute (ASA) indicates 32-bit addresses.

The "Immediate" field contains an operand which is an immediate part of the instruction itself. These operands may be one, two, or four bytes in length. An instruction which ordinarily calls for a 16-bit immediate operand will be assembled with a 32-bit operand if the operand size attribute (OSA) indicates 32-bit operands.

Segments which are assigned the the USE32 parameter have 32-bit address and operand attributes by default. Other segments have 16-bit attributes. An address size override prefix (ASO) reverses the prevailing address size attribute for the present instruction; whereas, an operand size override prefix (OSO) reverses the prevailing operand size attribute for the present instruction.

NOTE: Although Small Assembler will recognize the USE32 parameter in SEGMENT directives, the object files it produces are not compatible with LINK, and so are useless. When a standard 32-bit operating environment emerges, Small Assembler can be easily modified to conform to its object file requirements.

