Table of Contents

Notices

z/Architecture is a registered trademark of International Business Machines Corporation.

Introduction

Stand-alone mainframe bare metal programming needs are unique. They can be addressed by the GNU assembler, `as`, configured for s390 Executable and Linking Format (ELF) output, as is available with the Stand-alone Tool Kit (SATK).

GNU `as` itself has a number of annoying deficiencies for a mainframe assembler programmer used to proprietary assemblers.

The most notable are:

- no DSECT's,
- no USING and DROP statements,
- no cross reference listing,
- no auto alignment,
- no implied lengths,
- no backward ORG statement,

- limited macro language.

Bare metal programming really needs to create memory content images or other output formats tailored to the bare metal environment (for example, emulator or virtual machine). GNU tools can be coerced into doing this, but not without external support and tooling. SATK provides this tooling but the GNU `as` limitations persist.

GNU `as` can not be used as an integral tool for the creation of output (with the exception of GNU Compiler Collection). It can only accept an external source statement input file.

The z390 project has a much better assembler than GNU `as`, but for stand-alone programming lacks privileged instructions. However, it only produces, in this context, an object deck. A loader is required to use this output. Such loaders are readily available. However, in some contexts, a memory image or other format is required and z390 is not capable of creating other output formats. External tooling would be required to do so.

Proprietary assemblers suffer from the same limitation as z390 with regards to output formats.

Overview

A Small Mainframe Assembler (ASMA) addresses the bare-metal programming needs for mainframes. It has the look and feel of a traditional mainframe assembler. It does not suffer from the limitations of GNU `as`. Familiarity with mainframe assembler technology and instructions is expected of the reader.

ASMA supports two related but **separate** languages:

- an assembly language for mainframe machine instructions, and,
- a macro language for dynamic generation of assembly language statements.

See the “Assembly Language” and “Macro Language” sections for specifics on the ASMA assembly and macro languages and where they diverge from traditional mainframe assemblers.

Within this document certain descriptions will be initiated by one of these expressions in **bold face** font. They have the implied significance described here:

- **ASMA Limitation**: describes a capability typically available with mainframe assemblers but unavailable in ASMA.
- **ASMA Specific Behavior**: describes a capability that diverges from typical mainframe assemblers or is unique to ASMA and unavailable in those other assemblers.

To comprehensively document such unique aspects of ASMA is a major goal of this document. However, it is possible that some differences are missed. If in doubt, and you do not see something discussed here, assume it is not supported.

Descriptions identified as “ASMA Limitations” are likely candidates for removal in future versions of ASMA, while no commitment is implied to do so.

Before use of ASMA, the recommendation is strongly made that this document

be reviewed allowing the user to assess what modifications to existing code or the user's own coding practices may be required.

ASMA Limitation: ASMA does not have the concept of “open code”. Macro language directives are not recognized outside of macro definitions. Symbolic variables are only replaced when they occur within macro **model** statements. Conditional assembly is not supported outside of a macro. Wrap a legacy program that uses conditional assembly within a macro definition.

Invoking the Assembler

Two methods are supported for invoking of the assembler:

- from a command line or
- from within a Python script as an embedded assembler.

Initiating the assembler from a command line actually is an example of the second case. The command line script acts as a “wrapper” for the embedded use case of the assembler.

System Requirements

The assembler requires Python version 3.3 or later and will run on any platform on which the required level is installed. Install this version if it is not already available on the system intended to use ASMA. ASMA is available within and heavily dependent upon the tools provided by the Stand-alone Tool Kit, SATK. SATK is available from github:

https://github.com/s390guy/SATK

The Python directory search order is dynamically set by the tool by adding these directories:

```
${SATK_DIR}/tools/lang
${SATK_DIR}/tools/ipl
${SATK_DIR}/tools
```

`${SATK_DIR}` is the directory into which SATK is installed.

The `PYTHONPATH` environment variable is not normally required.

Environment Variables

Three search path environment variables consistent with the platform's conventions are used:

- `ASMPATH` – defines the search path for assembler `COPY` directives and the input file,
- `CDPGPATH` – defines the search path for a code page specification file, and
- `MSLPATH` – defines the search path for MSL `include` statements. If not defined, the `MSLPATH` defaults to the directory `${SATK_DIR}/asma/msl`.

The MSLPATH environment variable is not normally required.

By convention SATK ASMA source files use the .asm suffix, but are not required to do so.

One Python environment variable can effect performance: PYTHONOPTIMIZE. If set to a non-empty value, Python creates optimized Python byte code. This is equivalent to the Python command-line option -O and implied by the -OO option. Use of either this environment variable, or the command line option, may improve performance depending upon the size and content of the assembled source file. See the section “Python Exceptions” for more information.

Command Line Options

ASMA is initiated by a command line identifying the Python module asma.py. Refer to Python documentation for the mechanisms available on a specific platform for invoking a Python script and specifying the active PYTHONPATH environment variable.

Assembler command line options are specified as Python script file arguments. On some systems, the Python module itself can be specified directly on the command line:

```
asma.py [script file arguments]
```

Some systems may require first python itself be invoked on the command line:

```
python asma.py  [script file arguments]
```

or possibly

```
python3.3 asma.py [script file arguments]
```

Either form of command-line is supported by ASMA. On systems that support the first form, ASMA assumes that the path to the Python version 3.3 executable is:

```
/usr/bin/python3.3
```

Refer to Python platform specific documentation describing how **script file arguments** are provided on a command line. While the manner in which script file arguments are specified in a command line is platform dependent, the arguments themselves are platform independent. This section describes the script file arguments supported by asma.py.

For ASMA the script file arguments take the form of command line options with either a short or long form. Command line options with only a long form are not expected to be used under normal operation of the assembler. These options are used primarily for debugging purposes.

ASMA script file arguments fall into three broad categories:

- assembler controls,
- input options and
- output options.

A number of options specify a file. If a relative path or just the file name is specified, the file

must be accessible from the current directory search path or an identified path environment variable.

Assembler Controls

Assembler controls influence the operation of the assembler.

-t/--target ISA

The `-t` or `--target` option specifies the instruction set architecture targeted by the assembly. The following MSL file and CPU are implied by the following values supported for `--target`. If omitted, the argument defaults to `24`. The implied MSL file must be located within either the default MSL directory or the specified directory search order in the `MSLPATH` environment variable.

The primary attributes of each value and implied MSL file and CPU are provided in the following table.

CPU Architecture	`--target`	MSL File	MSL File CPU	Implied `--addr`	XMODE PSW	XMODE CCW
System/360	s360	s360-insn.msl	s360	24	360	CCW0
System/370	s370	s370-insn.msl	s370	24	EC	CCW0
Hercules s/380	s380	s380-insn.msl	s380	31	380	CCW0
370-XA	370xa	s370XA-insn.msl	s370XA	31	XA	CCW1
ESA/370	e370	e370-insn.msl	e370	31	E370	CCW1
ESA/390	e390	e390-insn.msl	e390	31	E390	CCW1
ESA/390 on z/Architecture	s390	s390x-insn.msl	s390	31	E390	CCW1
z/Architecture	s390x	s390x-insn.msl	s390x	64	Z	CCW1
all (see Note 1)	24	all-insn.msl	24	24	EC	CCW0
all (see Note 1)	31	all-insn.msl	31	31	E390	CCW0
all (see Note 1)	64	All-insn.msl	64	64	E390	CCW0

Note 1. All supported instructions are recognized by the assembler for this target architecture. The selection of `XMODE PSW` and `CCW` values are those valid for the IPL PSW and IPL CCW's for the environment implied by the target argument. The `64` target value uses the syntax of the `IPTE` instruction in z/Architecture while the `24` and `31` target values use the original `IPTE` instruction syntax.

--bltin

The `--bltin` option enables built-in macros. Presently only a simple built-in macro named `TEST` is implemented. Normally this option should be omitted causing built-in macros to be

disabled. This option is intended for possible future development.

--ccw FORMAT

The `--ccw` option sets the initial execution mode CCW, overriding the expected CCW defined for the CPU in the MSL database. Accepted values are:

`0`, `1`, and `none`.

A value of `none` removes the `XMODE CCW` setting, disabling the `CCW` directive.

--cp TRANS[=FILE]

Specifies the specific ASCII and EBCDIC character translation, `TRANS`, code page used by the assembler. If not specified, it defaults to the `94C` definition assumed to be available in the default code page file. Specify the optional `FILE` if a different code page file is used. See the "Code Page" section for details.

--cpu MSLFILE=CPU

The `--cpu` option identifies both the MSL file and the CPU within it for which the assembly is targeted. The two are specified separated without spaces by an intervening equal sign '=' as shown by this example:

```
--cpu s360-insn.msl=2020
```

-e/--error LEVEL

The `-e` or `--error` option specifies the level of error handling and method of reporting in use. Three values are supported:

- '`0`' – No error handling is provided. The assembler terminates immediately with a Python exception. Should only be used for diagnostic purposes.
- '`1`' – User errors are displayed to the user when detected by the assembler. Additionally, the error is provided in the listing with the statement in error and summarized at the end of the listing. Useful for problem isolation.
- '`2`' – User errors are provided in the listing with the statement in error and summarized at the end of the listing. Additionally the error summary is displayed to the user. This option is the default level of error handling and should be considered normal error handling.
- '`3`' – User errors are provided only in the listing with the statement in error and summarized at the end of the listing.

To the user, the apparent difference between options '`1`' and '`2`' is the sequence of displayed errors. When using option '`1`' errors may not be in assembly language statement sequence (because they are displayed when encountered). When using option '`2`', errors will be in

assembly language statement sequence (because they are sorted after all errors have been identified). On occasion the number of errors may be too large for practical use of displayed errors. Option '3' limits errors to the listing file only. Redirection of displayed output to a file may be required in some cases

-h/--help

The -h or --help option displays the script file options available and exits.

--nest DEPTH

The -n or --nest option sets the maximum number of nested input sources. An input source may be a file, or a macro expansion.

--psw FORMAT

The --psw option sets the initial execution mode PSW format, overriding the expected PSW format defined for the CPU in the MSL database. Accepted values are:

S, 360, 67, BC, EC, 380, XA, E370, E390, Z, and none.

A value of none removes the XMODE PSW setting, disabling the PSW directive.

Input Options

Input options identify for the assembler the source of input.

source

The source positional option identifies the input assembler source text file being assembled. It must be specified as the last script file argument and is required. Whatever is encountered as the last option is treated as the source text file with or without a path.

If a relative path or file name alone is used, the ASMPATH directory search order is used to locate the file as with a COPY directive.

--case

Enables case-sensitive handling of assembler language labels, macro language symbolic variables and macro language sequence symbols. By default input is case-insensitive.

-D SYMBOL[=VALUE]

The -D option establishes a global character symbolic variable and, if provided, assigns it a character string value. Supplying only a symbol is equivalent to a GBLC macro language directive. Providing a value is equivalent to the GBLC being followed by a SETC macro language directive.

This option performs a role similar to the system symbolic variable &SYSPARM but allows any symbol other than those already utilized by ASMA as system symbolic variables to be specified. Unlike the &SYSPARM variable, symbols defined by the `-D` option are normal read-write symbols and may be altered by a defined macro. As with all user defined symbolic variables, a macro must declare its usage before accessing the symbol.

Output Options

Output options influence the output created by the assembler.

-d/--dump

The `-d` or `--dump` option cause the assembly listing, if itself is not suppressed, to include the image file to be included in the format of a storage dump. Both hexadecimal object and ASCII and EBCDIC characters are interpreted in the dump. Memory addresses bound to each region and positions of the image content relative to the start of the binary image are both provided in the dump.

-h/--help

The `-h` or `--help` option displays the script file options available and exits.

-a/--addr SIZE

The `-a` or `--addr` option overrides in the listing, the address size specified within the MSL for the target CPU. Only four values are accepted: 16, 24, 31, or 64. The option applies to the statement area of the listing and the optional image file dump.

-g/--gldipl FILEPATH

The `-g` or `--gldipl` option specifies a the file to which a generic list directed IPL file is written. Files participating in the list directed IPL are written to the same directory. If not specified, all list directed IPL files are suppressed. See the “Using List Directed IPL” section for details.

-i/--image FILEPATH

The `-i` or `--image` option specifies the file to which the binary image is written. If not specified, the image file is suppressed. See the “Using the Image File” section for details.

-l/--listing FILEPATH

The `-l` (lower-case L) or `--listing` option specifies the file to which the assembly listing is written. If not specified, the assembly listing is suppressed. Familiarity with mainframe assembler listings is expected to make the structure and content of the listing self-explanatory.

-o/--object FILEPATH

The `-o` pr `--object` option identifies the file to which a loadable object deck is written. If the option is not specified, an object deck is not created. See the “Using the Object Deck” section for details.

-r/--rc FILEPATH

The `-r` or `--rc` option identifies the file to which a Hercules RC script file containing real storage alter commands is written. If the option is not specified, the RC script file is not created. See the “Using the RC Script File” section for details.

-s/--store FILEPATH

The `-s` or `--store` option identifies the file to which a series of real storage STORE commands is written. If the option is not specified, the command file is not created. See the “Using a STORE Command File” section for details.

-v/--vmc FILEPATH

The `-v` or `--vmc` option identifies the file to which a series of real storage STORE commands is written. If the option is not specified, the command file is not created. See the “Using a STORE Command File” section for details.

--debug OPTION , --oper OPER, --pas N

These three options control various forms of debug output. Under normal circumstances, they should not be used. The `--help` option lists available values. Debug output may be limited or non-existent if the Python environment variable `PYTHONOPTIMIZE` or Python command line options `-O` or `-OO` are used.

ASMA Assembly Listing

ASMA assembly listings contain as many as seven distinct reports depending upon the assembly and command line options. The entire listing is controlled by the `--listing` command-line script argument. If omitted, no listing is produced.

The following table identifies the individual reports and the controls that influence the generation of the report within the listing. Reports for which no control is provided are always produced if the listing itself is generated.

Report	Control	Description
1	none	Assembled statements and their generated binary content
2	none	Symbol table information and symbol cross-reference
3	none	Macro cross-reference listing

Report	Control	Description
4	none	Image map reporting image and memory residency of the image, each region within the image and each control section within each region.
5	`--dump`	Image content in the format of a memory dump if `--image` also specified.
6	`--dump`	Generated deck records if option `--object` also specified.
7	none	List of referenced files by number. File numbers are reported in the error report.
8	none	Error report. May contain only informational messages or no errors when none encountered.

Other forms of output are all text files and are not added to the listing. Direct inspection of the generated files is required to see the content.

Code Pages

ASMA input is **ASCII** based. Handling of input text is controlled by the Python support of Unicode. All input text files are treated by Python as UTF-8 character encoding as its implemented by Python. Support for a different input encoding, while supported by Python, is not presently available to an ASMA user.

ASMA Limitation: The legacy practice of enclosing arbitrary values within a character self-defining term is inhibited by Python's own text handling. Such character sequences must be replaced by either a decimal, hexadecimal or binary self-defining term.

When assembling character constants or referencing character symbolic variables in macro directives, the assembler must know how to convert ASCII characters presented to it into EBCDIC characters. Conversely when interpreting binary data as characters, the assembler must know the character represented by the binary data. The `codepage.py` module provides this support.

`codepage.py` provides both the internal support for code page usage and a command line interface for displaying useful information about code page definitions. Code page definitions follow the same general coding rules as are used with MSL files. A set of built-in definitions constitute the default character sets and translation options. Each translation definition utilizes an ASCII and EBCDIC code page definition to construct three translation tables:

1. ASCII-to-EBCDIC translation,
2. EBCDIC-to-ASCII translation, and
3. Binary interpretation translation.

ASCII/EBCDIC translation tables convert between the two code sets based upon those

characters **shared** between them. Unshared or undefined code assignments are not translated.

Binary interpretation translates all single byte binary values to an ASCII character. Four approaches may be used for these conversions when an assignment exists for a code point:

1. Interpret only ASCII codes from the ASCII code page,
2. Interpret only EBCDIC codes from the EBCDIC code page,
3. Interpret both ASCII and EBCDIC codes from their respective code pages using the ASCII code assignment when a code assignment exists in both code pages, or
4. Interpret both ASCII and EBCDIC codes from their respective code pages using the EBCDIC code assignment when a code assignment exists in both code pages.

Unassigned values utilize a default “fill” character specified within the definition. The default definitions use an ASCII period, '`.`', as the fill character.

Built-In Code Page Definitions

The built-in definitions identify a single set of characters. These are the characters specified by an architecture reference summary manual as the '94C' assignments in its “Code Assignments” section with the exception of four characters:

- E0, the Eight Ones character;
- NSP, the Numeric Space character;
- RSP, the Required Space character; and
- SHY, the Soft Hyphen character.

The built-in definitions provide the four translation options using the identification:

- `94C` – Interpret EBCDIC only (approach 2 above),
- `94Ca` – Interpret ASCII only (approach 1 above),
- `94Cea` – Interpret ASCII and EBCDIC with EBCDIC preferred over ASCII (approach 4 above), and
- `94Cae` – Interpret ASCII and EBCDIC with ASCII preferred over EBCDIC (approach 3 above).

Any of these translation definitions may be used in the `--cptrans` command line option. `94C` is the default.

It is possible when using the set of 94C definitions to generate unrecognized ASCII characters on a typical PC keyboard. These will appear in assembled object code and macro language character symbolic variables as unchanged.

Local Code Page Modifications

The `codepage.py` module was designed specifically with local modifications in mind. The built-in definitions can be output to a file using the `codepage.py --write` command line option. Once output, the file may be locally modified to satisfy local code page needs. The local file and translation code page is then referenced in the ASMA `--cp` command line option for use by the assembler.

Two character set groups are provided within the default definitions for local changes: EBCDIC-local and ASCII-local. They appear at the beginning of the file. Apply your local definitions there. Because duplicate definitions are not allowed, in some cases a supplied character definition may need to be turned into a comment.

The `codepage.py` command line options, described in the next section, are provided to assist in verifying the expected results.

Changing the supplied default definitions from within the `codepage.py` module is also possible but not recommended.

codepage.py Command-Line Options

-a/--a2e

If present, print the ASCII-to-EBCDIC translation table of the selected translation ID as a 16-by-16 matrix.

-c/--codepages

If present, print the ASCII and EBCDIC code pages of the selected translation ID as a list sorted by character name.

-e/--e2a

If present, print the EBCDIC-to-ASCII translation table of the selected translation ID as a 16-by-16 matrix.

-p/--codepoints

If present, print the list of ASCII and EBCDIC characters, by name, assigned to each code point.

--cpfile

Specifies the code page file defining the ASCII and EBCDIC character translations available to the command line utility. The `CDPGPATH` environment variable defines the directory search path used to locate the file. If not specified the built-in translation definitions are used.

--cptrans

Specified the specific ASCII and EBCDIC character translation definition upon with the output options apply. If omitted the id `94C` is used.

--dumpa

If present, print the binary interpretation table of the selected translation ID as 16-by-16 matrix. Translation is to ASCII output characters.

--dumpe

If present, print the binary interpretation table of the selected translation ID as 16-by-16 matrix. Translation is to EBCDIC output characters.

--test

Perform a simple translation test on the value supplied with the argument.

Troubleshooting

When unexpected problems occur take the following steps:

Do not set the Python `PYTHONOPTIMIZE` environment variable. This option causes the removal of internal value checking and suppression of some debug or tracing results. Creation of optimized byte-code removes assert statements and others operating under control of the Python variable `__debug__`. Normally these are not needed and assembly times are improved.

Modify the `--error` reporting level to improve error reporting. The default, `2`, causes errors to be reported at the end of the assembly and in the listing. Unanticipated Python exceptions will cause the assembler to terminate before a listing is produced and errors are reported. Use an error level of `1` to have errors reported when they are detected. Use a level of `0` to disable Python exception handling. This may result in the initial detected error being displayed.

If the failing statement can be identified, use the `--oper` to trace processing of a specific operation. More general processing can be enabled by using `--debug`. Both of these options can generate excessive output. Output from these options are not documented and is only meaningful if the related Python code is understood. If these two options are used, it is recommended to create a test file that contains the problem statement or statements to reduce the output.

If you remove the definition for the `PYTHONOPTIMIZE` environment variable but debugging or tracing information is not appearing, ensure any directories named `__pycache__` are removed. This directory is where Python stores compiled byte-code. If the cached byte-code has removed optional error checking code, the directory's removal will force recreation of the Python byte-code with the error checking code present.

Python Exceptions

Generally any Python exception raised by the assembler is a bug.

The exception to this statement is when the command line option `--error 0` is in effect. This option disables all internal trapping of Python exceptions. When this option is in use, `AssemblerError` exceptions should be considered as normal. These exceptions are used when a user error is encountered. All other exceptions are bugs.

There are occasions where the reason a statement is reported in error is not entirely clear. This can result from the trapping of other exceptions that become translated into an `AssemblerError` exception. By using command line `--error 0` option the triggering exception may be reported. In this case, the exact location of the inner error will be reported facilitating resolution.

Regardless of the `--error` option in effect, a `ValueError` or `AssertionError` exception is a bug. Internal checks that result in unanticipated conditions raise a `ValueError` or `AssertionError` exception. They are never caught and always immediately terminate the assembler. Python itself may raise a `ValueError` or other exception. Uncaught exceptions raised by either Python or the assembler always terminate the assembler and always require correction.

Legacy Programs

This section brings together in a single place some of the areas of adjustment that are required by ASMA to an existing assembly source program. Refer to details within the document for more information.

Most programs accepted by a proprietary assembler will likely require some modification for ASMA.

The following sections are based upon limited experience in using ASMA with legacy programs, but the areas discussed either reoccur or deserve explicit mention.

Although obvious, any assembler directives or constant types not supported by ASMA must be addressed.

Statement Format

ASMA expects variable length input lines. Generally the presence of sequence columns become interpreted as comments. Python supports a universal newline strategy which makes ASMA insensitive to the underlying platform's standard line termination sequence.

Comment statements start with a asterisk, `*`. If the asterisk is preceded by spaces or other “white space” characters such as tabs, the statement will be found in error.

ASMA expects continuation lines to be indicated by a backslash, `\`, in the last character position of the variable-length line.

Only in the case where a legacy program utilizes both sequence columns and has a continuation in column 72 will ASMA not recognize the continuation. Plans exist to support 80-column input records in the legacy format but is not yet available. For such statements the sequence columns must be removed and the continuation character in column 72 must be changed to a backslash character.

If problems result with continuation lines, remove continuation by creating a single long line.

The alternate continuation conventions available with legacy macros is not supported by ASMA. In this case all comments and spaces between the final comma and the continuation character must be removed.

LTORG Directive

Literal pools are not supported by ASMA. Literals used in instructions must be replaced by an explicit `DC` directive.

Example:

```
LA    1,=F'1'
```

```
            LTORG
```

becomes:

```
            LA    1,LIT1
LIT1        DC    F'1'
```

D-type Constants

A common legacy practice is to utilize a `D`-type constant for doubleword alignment and placement of binary zeros into an 8-byte field. ASMA treats `D`-type constants as a synonym for the `FD`-type constant. Floating point nominal value syntax is not supported. Only integer nominal values are recognized.

Floating-Point Constants

ASMA lacks support for all floating point constant types. Although floating point instructions are supported, the creation of constants for them is not. The only available solution is for such constants to be replaced by `X`-type constants. An assembly listing from another source could provide the required content. Because use of floating-point instructions is typically rare for a bare-metal program, it is not expected to be a major practical limitation in this context. Resolution of this limitation is an eventual goal.

Vector Instructions

Vector instructions are not presently supported by ASMA. If the reader has a need for vector instructions the reader is encouraged to work with the author in developing their support within ASMA. MSL is the path to such support.

Image Structure

ASMA Specific Behavior: Everything in this section is unique to ASMA or diverges from legacy practices.

The output of the ASMA assembler is not the typical object deck composed of ESD, TXT, etc. records. ASMA creates a binary image file. Conceptually it is something one might burn into a ROM on more modern systems. It is one large “chunk” of binary data. From an external point of view the image file has no structure. The program in the image can impose its own structure on the image content through the use of the assembly directives `START`, `REGION` and `CSECT`.

To illustrate, an image could contain an installer, when executed, installs another program (also part of the image) on a device that itself could be used as the IPL source for the installed program. Some object formats, for example, an Executable and Linking Format (ELF) executable file, requires external tooling to accomplish the same task. Such tooling is available within the SATK through use of the `iplmed.py` utility. Such tooling limits the creation of IPL media to the ELF's abilities and standards. The image file approach offers much greater flexibility in this area. The image structuring directives allow the program to recognize and specify the residency requirements of different portions of the image.

Regions

A region contains binary content bound to a starting memory address identified in the `START` directive that initiates the region's creation. Unlike typical mainframe assemblers, multiple `START` directives are possible allowing the creation of multiple “regions”. Regions are placed contiguously within the image in the order in which the `START` directives that initiate them are encountered. A named region is identified by the `region` operand of its associated `START` directive. An unnamed region is created when the associated `START` directive has the `region` operand omitted. A new `START` directive causes the following content to be placed within the newly initiated region and newly created control section. Content within a previously created region can be continued by use of a `REGION` directive.

Control Sections

Each region is populated by one or more control sections each of which is identified and initiated by a `CSECT` or `START` directive. A new control section is placed within the currently active region under construction with double word alignment. Like regions, a new control section will cause all succeeding content to be placed in the new control section. Content can again be placed in the previous control section by coding a `CSECT` directive with the label field containing the name of the control section being continued or omitted when the unnamed control section is targeted. Portions of a control section are made contiguous within the control section even if the assembler statements creating the content are not contiguous.

Control sections can not be created outside of a region. Once initiated within a region, all of the control section's content will be placed within the region. By continuing a control section, automatically the control section's region becomes active. Correspondingly, when a region is continued by a `REGION` directive, the previously interrupted control section becomes the control section into which new content is placed. A succeeding `CSECT` directive will change the active control section to another and to its associated region.

Dummy Sections

Dummy sections are not associated with any region and are never explicitly bound to a memory address. Only section relative addresses are ever assigned to symbols within a dummy section.

Image, Region and Control Section Attributes

The image, each named region and each named control section are identified by a symbol. Each symbol has certain attributes accessible from within the program. The symbol associated with a control section or region is specified in the label field of the respective initiating `CSECT` or `START` directive. The image is automatically assigned the name `IMAGE` in all upper case. By its early definition and the restriction on duplicate symbols, this name is really a reserved symbol. The only such symbol in the system.

All symbols have associated with them a value which may be an address or an integer value. In the case of the symbols used for an image, region or control section, the value is the starting absolute address as bound at the completion of Pass 1. The address assigned to the image is the address assigned in the first `START` directive of the program. This address is effectively the address to which the image expects to be placed in memory. The length attribute, referenced by preceding a symbol with the sequence `L'`, is the total length in bytes of the image, region or control section.

Unlike other symbols, each image, region and control section symbol has an additional attribute, the “image” attribute. Similar to the length attribute, it is referenced by preceding the symbol with the sequence `I'`. The image attribute provides the displacement from the start of the image of the corresponding symbol. Through this attribute a program can locate a region or control section within the image independent of its bound memory address.

The unnamed region's or unnamed control section's attributes are not accessible to the program because a symbol is not associated with either. Only named region or named control section attributes are accessible to the program via the assigned symbol.

Using the Image File

Making use of the image file requires it to have been created in a way that makes it useful. The image file contains no structure discernible externally. The image file is anticipated to be loaded into memory at the address specified in the first `START` directive. If the expectation is that the image file will be placed into memory and that a restart interruption will be used to

enter it for execution, then the image file must be constructed with this in mind. If the image file is intended to contain the Restart New PSW, the image file must expect to be loaded at the memory address of the Restart New PSW. For systems not in z/Architecture ® mode, the assigned storage location of the Restart New PSW is address 0. For systems in z/Architecture mode, the assigned storage location is address 288, or in hexadecimal, 0x120. This drives the operand of the `START` directive to be either `X'0'` or `X'120'` (or if decimal is preferred, `0` or `288`). At the location of the Restart New PSW within the image file, must be a properly formatted PSW that will initiate running of the program within the image.

An image file constructed this way could also utilize list directed IPL. However, because the system will not be in z/Architecture mode when the IPL PSW takes control, the starting location of the first region must be 0 at which location must be a PSW generated by the assembly. Supplying the content of the list used during the IPL process is the responsibility of the user of the image file. See the section “Using List Directed IPL” for more information.

Once loaded into storage, it becomes the responsibility of the loaded program to properly locate within storage any regions other than the first contained in the image. The region or control section symbol `I'` and `L'` attributes are intended to assist the loaded program in performing this content relocation.

Hercules `loadcore` command can load an image file into emulated main storage. Additional commands related specifically to the loaded image file are required to start its execution. The next section, “Using List Directed IPL”, provides a more error free method that will directly execute the loaded image.

Using List Directed IPL

List directed IPL allows one of more files to be loaded into storage with control passed to the loaded content by causing the IPL PSW at address 0 to become the active PSW. The content of the directory is oriented towards use by the Hercules `ipl` console command with a file path. Refer to Hercules documentation for details. Multiple platforms may support list directed IPL. It is the user's responsibility to position the list directed IPL files in a compatible location and identification when using a platform other than Hercules.

The `--gldipl` option specifies the file into which IPL list is written. This option writes a file to the same directory as the IPL list file for each **region** defined in the assembly. Each region file uses the region's name as specified from its `START` directive and file extension of `.bin`. The unnamed region may not be used with list directed IPL. The IPL list file has the name of the image with the file extension provided in the command line option. It is the IPL list file that is specified in the Hercules `ipl` command.

Because the IPL function automatically uses the IPL PSW at absolute storage location 0, an assembly intended for use with list directed IPL must ensure a PSW is placed in storage at address 0 by the loaded regions. The PSW may be the only content of the region starting at address 0 or part of additional region content.

To illustrate this structure, assume an assembly contains two regions. The `IPLPSW` region

starts at address 0 and only contains a 64-bit PSW. The main program region is `PROGRAM` and starts at location `X'2000'`.

The following example uses a UNIX-like environment, but ASMA runs on any platform supported by Python with python tailoring file names to the local environment. The user `johndoe` uses the following command line option in the assembly:

```
--gldipl test/program/test.ipl
```

All of the list directed IPL files will be written to the `/home/johndoe/test/program` directory. A relative directory is expanded to an absolute path. Three files will be placed in this directory:

`test.ipl` – the IPL list used during the IPL function

`IPLPSW.bin` – the 8-byte PSW

`PROGRAM.bin` – the actual program

The IPL list file (`test.ipl`) will contain two lines of text, as follows:

```
PROGRAM 0x2000
IPLPSW 0x0
```

The sequence of regions in the IPL list file, and hence the sequence in which regions are loaded into memory, is dictated by the sequence in which the regions are defined in the assembly.

The actual IPL command used with Hercules, likely placed in a RC script file or the Hercules configuration file, would be:

```
ipl test/program/test.ipl
```

The Hercules list directed IPL command does not support absolute paths. This statement assumes the current working directory is `/home/johndoe`.

Using the Object Deck

An object deck is created only when the `--object` command line option is specified. The deck contains one or more TXT records and an END record. TXT and END records are limited to three-byte address fields. If the assembly contains object code at any location with an address larger than `X'FFFFFF'`, the object deck is suppressed with a warning message. The END record contains the entry point identified by the last `ENTRY` directive. If no `ENTRY` directive is present in the assembly, the object deck is suppressed.

TXT records will only contain object code generated by assembly machine instructions or directives. Other than the address of the TXT record content, regions and control sections have no other influence on the output. Areas not explicitly initialized are not contained in the TXT record binary content. This means that areas defined by `DS` directives or gaps created due to implied alignments are not filled with binary zeros as is the case with the binary image file. Because control is passed to the loaded content by an object deck loader, an explicit

PSW in storage is not required for execution entry to the TXT content. A separate card-based loader that may be the target of an Initial Program Load function is required and, if used, must precede the actual object deck card images created by this option.

The Hercules `loadtext` console command may also be use to the load the content into storage. Unlike the card-based loader, a Restart New PSW would be required to manually pass control to the loaded program. A manual, or script driven, restart interruption would be required to actually cause control to be given to the program.

The object deck created by the `--object` option is not suitable for use with a linkage editor. It is only usable with a loader facility supporting TXT and END records.

Using the RC Script File

The Hercules emulator allows storage to be loaded using a real storage alter command, the Hercules `r` console command. The `--rc` option creates a script file containing a series of real storage alter commands that have the effect of loading the assembly content. As with the object deck, only areas explicitly initialized are contained in the real storage alter commands. Areas defined by `DS` directives or gaps created due to implied alignments do not cause real storage to be altered. Other than supplying the address of the storage alter command, regions and control sections have no other influence on the output. Each command takes the form of:

```
r address=hexdata
```

Because Hercules allows a script file to include other script files, the file created by the `--rc` option may be directly included in another file.

Depending upon how the user of the script file expects to pass control to the loaded storage content, a Restart New PSW may be required at real storage address 0 (`X'0'`) or 288 (`X'120'`). When using a script file, the easiest way to achieve this is by creating a region with the starting address of 0 or 288 into which is assembled the Restart New PSW. Other mechanisms exist for starting the CPU executing with Hercules console commands. An assembly generated Restart New PSW is just one way to achieve this without more error prone mechanisms.

Using a STORE Command File

A STORE command file contains a series of STORE commands that alter real storage content using a mainframe virtual machine environment. As with the object deck, only areas explicitly initialized are contained in the real storage alter commands. Areas defined by `DS` directives or gaps created due to implied alignments do not cause real storage to be altered. Other than supplying the address of the storage alter command, regions and control sections have no other influence on the output.

Two options create a STORE command file: `--store` and `-vmc`. The only difference between the two files is that the output from the `--vmc` option expects to be executed from a

mainframe virtual machine environment. Each command within the `--vmc` generated STORE command file starts with `CP`. Output from the `--store` option does not contain the `CP` sequence. Each command takes the form of:

`STORE R S address hexdata` (for output from the `--store` option), or

`CP STORE R S address hexdata` (for output from the `--vmc` option).

Depending upon how the user of the STORE command file expects to pass control to the loaded storage content, a Restart New PSW may be required at real storage address 0 (`X'0'`) or 288 (`X'120'`). When using a STORE command file, the easiest way to achieve this is by creating a region with the starting address of 0 or 288 into which is assembled the Restart New PSW. Other mechanisms may exist for starting the CPU within a virtual machine environment. An assembly generated Restart New PSW is just one way to achieve this without more error prone mechanisms.

It is the user's responsibility to move the contents of the STORE command file into the virtual machine environment for actual execution of the real storage altering commands.

Assembly Language

ASMA performs three passes during its processing. An error can cause immediate failure or all errors can be reported at the end of the assembly depending upon the command line options selected.

ASMA Limitation: The concept of continuation lines are not supported by ASMA.

Pass 0

All text statements enter the assembler in Pass 0.

Empty lines and comment lines are recognized with no further processing.

Statements are broken up into fields: label, operation, and operands with comments.

Statements containing an operation field for the following directives are processed in this pass: `COPY`, `MACRO`, `MHELP`, `PRINT`, `SPACE`, `TITLE` and `XMODE`.

Macro definitions are created and macro expansions are invoked. See the “Macro Language” section for details.

Other statements are analyzed as follows:

1. Symbolic variable substitution is performed. See the “Model Statements” section for details.
2. Assembler directives and machine instructions are recognized and execution mode options applied.
3. Parsing of operands into expressions as required by the identified operation. Expressions are not evaluated at this time, just parsed. Calculating the value of expressions occurs in either Pass 1 or Pass 2 depending upon the directive.
4. Comments on a statement are ignored.

Source statements brought into the input stream by the `COPY` directive occurs in this pass.

At the completion of this pass, no new statements can be added to the assembly process and all statements without errors are processed by the next pass.

Pass 1

For instructions and other content producing directives, their bound locations are established and the size of their content is determined, although the content itself is

not.

For labels, they are assigned a bound address of the current location of the current active control section.

For assembler directives not producing image content, they are acted upon at this point:

CSECT, DS, DSECT, EJECT, END, EQU, ORG, REGION, START

Calculations performed by any of these directives require referenced symbols to already be defined, namely EQU, ORG and START.

At the end of Pass 1, the size of each statement's content and the size of each control section has been established. All statements without errors are processed by the next pass.

Pass 2

Each control section without an explicit starting location is established based upon its alignment attribute, its sequence within the assembly and its preceding control section's ending address.

Each symbol is bound to its final address based upon its position within its control section. The section “linking” process is now complete.

Content is created by evaluating all remaining expressions in the directives and processing the instruction. Content creating and content influencing statements of the assembler are completed during this pass:

CCW, CCW0, CCW1, DC, DROP, PSWS, PSW360, PSW67, PSWBC, PSWEC, PSWXA, PSWE370, PSWE390, PSWZ, USING, all machine instructions.

At the completion of Pass 2, all requested output formats are created. Each statement's content is placed in its location within its control section at its real location. See the various sections on using each of the various output formats.

ASMA Specific Behavior: The above description of ASMA's processing has implications for support of legacy type macros. One of the major features of the legacy macro language is its ability to access symbol attributes and influence instruction generation based upon them. Currently ASMA does not assign symbols until Pass 1. However, Pass 1 does not support the addition of statements into the input stream, the input stream having been completed in Pass 0 where macro's are expanded.

ASMA Limitation: ASMA does not have the concept of “open code”. Macro directives are not recognized outside of macro definitions. Symbolic variables are only replaced when they occur within macro model statements. Conditional assembly is not supported outside of a macro. Wrap a legacy program that uses conditional assembly within a macro definition.

ASMA Limitation: Lookahead mode is not supported. Conditional assembly statements AIF or AGO are not supported in “open code.” “Open code” must be wrapped within a macro definition for these statements to be effective.

ASMA Limitation: Expressions in assembler statements are limited to arithmetic computations: addition, subtraction, multiplication and division. Logical operators, for example, “and”, “or” and comparisons, while supported by the macro language, are not supported by the assembler.

MSL Considerations

ASMA machine instruction recognition and construction is completely driven by the MSL file selected by the `--target` and the `--cpu` command line options. Only machine instructions defined for the selected CPU in the MSL file will be recognized by ASMA.

ASMA Limitation: ASMA has no ability to support optional instruction operands. If an assembler instruction operand is optional, it will become required in the MSL and any source statements omitting the operand will fail assembly.

This limitation only explicitly impacts a few modern floating point instructions using the RRF format. These have optional assembler operands. When used within ASMA, the optional operand must be coded with a value of zero.

This limitation also influences MSL instruction definition and selection. For example, some legacy instructions are classified as storage-immediate. However, the immediate operand is normally not coded because it is ignored by the instruction. Omitting the operand in legacy assemblers causes the immediate field to contain binary zeros. For these instructions a different format in MSL is used. The `LOAD PSW` instruction is an example of this. Modern usage identifies a different format, the storage format for `LOAD PSW`.

ASMA Input Statements

ASMA input statements map into five elements in sequence as illustrated by this model:

```
LABEL   OPERATION  OPERANDS       COMMENTS    \
```

The only required element is the `OPERATION` field. `COMMENTS` are always optional. Whether a `LABEL` or one or more `OPERANDS` are required or optional is dictated by the `OPERATION` field content. If the statement is continued the last character immediately preceding the end-of-line sequence must be a backslash, `\`. If `OPERANDS` are not defined for an operation, they are considered to be comments. If a `LABEL` is omitted or in the rare case prohibited, the required `OPERATION` field must be preceded by at least one space. If a `LABEL` is present it must begin in the statement.

A statement following a continued statement must contain a space in positions 1-15. A continued statement may itself be continued. Content in the continued statement beginning with position 16 logically continues the `OPERANDS` of the preceding statement.

Two exceptions to this format are supported for comment statements. If the statement begins with either an asterisk, `*`, or a period followed by an asterisk, `.*`, the entire statement is free-form. Comment statements do not recognize continuation. A comment statement starting with a period in a macro definition are considered silent comments and are not treated as

model statements.

Names

Names are used within ASMA in various contexts. ASMA recognizes four different forms of names:

- Label symbol,
- Symbolic variable,
- Sequence symbols, and
- Label symbol terminated with a symbolic variable.

Labels

A label begins with any alphabetic character in upper or lower case ('A' through 'Z' and 'a' through 'z'), or any of three special characters ('$', '@' or underscore, '_') optionally followed by any of these characters or a number '0' through '9'. Labels may not begin with a number.

Labels in the LABEL field define symbols as address locations or specific values.

Labels in the OPERATION field identify machine instruction mnemonics, assembler or macro directives, or previously defined macros.

A label in the OPERAND field references a label defined in the LABEL field.

Symbolic Variables

Symbolic variables are labels preceded by an ampersand, '&'. Symbolic variables may only be used in macro language statements within a macro definition. Within macro model statements, symbolic variables may occur alone or following a label within the model's LABEL or OPERATION field and anywhere in the model statement's OPERAND field. Refer to the Macro Language section for other uses of symbolic variables.

ASMA Limitation: Symbolic variables will cause an error if occurring in Assembly Language statements.

Sequence Symbols

Sequence symbols are labels preceded by a period, '.'. Sequence symbols may only occur in Macro Language directives in the LABEL field or referenced in specific directives' OPERAND field.

ASMA Limitation: Sequence symbols may not be used outside of macro definitions.

Self-Defining Terms

ASMA input statements may contain self-defining terms. Self-defining terms define unsigned integers used alone or within expressions.

Self-defining terms look like assembler `DC` directive operands but function differently. They are utilized to define values within the assembly itself and never inherently define storage or address locations. They may be used in contexts that do.

Self-defining terms when used as the default value of a prototype keyword parameter result in a the string used to define the term as the keywords parameter value.

Binary

A binary self-defining term starts the an upper or lower case '`B`', followed by a single quote, a value composed of only a zero, '`0`' or one, '`1`' and ending with a single quote.

Example: `B'01001'` – value 9

Character

A character self-defining term starts with an upper or lower case '`C`', '`CA`' or '`CE`', followed by a single quote, a value composed of a **single** character and ending with a single quote. The value is defined by the ASCII or EBCDIC code point of the character value. A '`C`' implied the EBCDID code point.

Example: `C'0'` – value 240 or hexadecimal 0xF0.

Example: `CA'0'` – value 48 or hexadecimal 0x30.

Decimal

A decimal self-defining term is composed of a sequence of one or more numbers '`0`' through '`9`'. The term must not start with a sign.

Example: `920` – value 920 or hexadecimal 0x398.

The presence of a sign in conjunction with a self-defining term implies an expression and is only valid in contexts supporting expressions.

Hexadecimal

A hexadecimal self-defining term is composed of an upper or lower case X, followed by a single quote, a value of one or more hexadecimal digits, '`0`' through '`9`', '`A`' through '`F`' or '`a`' through '`f`' and ending with a single quote.

Example: `X'0Bad'` – value 2,989.

Quoted Strings

Quoted strings are surrounded by single quotation marks. Within quoted strings an ampersand or a single quote require two consecutive ampersands or single quotes to result in a single quote or ampersand within the quoted string. Quoted strings may be of any length. Quoted strings may be used in `DC` assembler directives for `C`, `CA` or `CE` type constants, default values assigned to keywords by a macro prototype statement or parameter values assigned when a macro is invoked.

Case Sensitivity

By default the assembler is case insensitive.

The command line argument `--case` enables treatment of assembler labels, macro symbolic variables and sequence symbols with case sensitivity. All other input remains case insensitive.

Assembler statement operation field is always case insensitive. Storage allocation and defined constant types and an associated length modifier are always case insensitive.

Depending upon the use or absence of the `--case` argument, values may occur appear as the are is statement input or with strictly upper case letters. Internally case insensitive data is converted to upper case. This is most obvious within the symbol cross-reference portion of the assembler listing. When case sensitivity is disabled, all symbols are listed in upper-case.

Operand Values

Operand values may be either an integer or bound to an address. All operand values are derived from a mathematical expression. The expression may be simply a numeric value or derived from a calculation.

Calculations between values are allowed as follows:

Operation	Left operand	Right operand	Result
+ (addition)	integer	integer	integer
+	integer	address	address
+	address	address	prohibited
+	address	Integer (Note 4)	address
- (subtraction)	integer	integer	integer
-	integer	address	prohibited
-	address	address	integer (Note 1)
-	address	Integer	address
* (multiplication)	integer	integer	integer

Operation	Left operand	Right operand	Result
*	integer	address	prohibited
*	address	address	prohibited
*	address	integer	prohibited
/ (division)	integer	Integer (Note 3)	integer (Note 2)
/	integer	address	prohibited
/	address	address	prohibited
/	address	integer	prohibited

Note 1: Subtraction only allowed between addresses of the same section.

Note 2: Remainder from a division operation is discarded.

Note 3: Division by zero results in a value of zero.

Note 4: **ASMA Specific Behavior**: Dummy section relative addresses are treated as integers in this context. The integer value is the symbol's location within the dummy section.

ASMA Specific Behavior: Expression results and intermediate values are not limited to values within the range -2^{31} to $+2^{31}-1$.

Location Counters

ASMA Specific Behavior: Everything in this section dealing with location counters is unique to ASMA. ASMA location counter management is both similar and dissimilar to all other legacy and modern mainframe assemblers.

Similar to legacy and dissimilar to modern mainframe assemblers, the user is unable to explicitly manage location counters. Similar to modern assemblers and dissimilar to legacy mainframe assemblers, multiple location counters are used. Whenever a control section or dummy section is initiated it is assigned its own location counter. During pass 1, section relative addresses are utilized. During pass 2, control section relative addresses are assigned an absolute address based upon the region's starting absolute address and the location of the control section within the region. This occurs during the implicit linking process at the end of Pass 2 when the stand-alone image is created.

This processing of location counters has the effect of altering what one would find in a legacy mainframe assembler address column within the listing. Because the listing is produced based upon the control section's absolute region address, it may be logically contiguous with the preceding control section or not. By managing region start addresses and control section assignments, address management may be managed largely similar to that of a modern mainframe assembler providing location counter related assembler directives.

ASMA location counter management is most similar to the GNU `as` assembler, although the GNU `as` assembler does not allow absolute address assignment to control sections. That

process is managed during linkage processing by the GNU `ld` linkage editor via linkage editor scripts, albeit only within the context of the creation of Executable and Linking Format (ELF) executable files.

Machine Instructions

Instruction recognition is defined external to the assembler by means of a text file specified explicitly in the command line or implicitly by the targeted architecture. The file is coded using the Machine Specification Language (MSL). Only central processing units (CPU) defined in the file can be a target for the assembler. Only instructions identified as valid for a target CPU by the MSL file are recognized by the assembler.

Assembler Directives

ASMA assembler directives are documented here. The following conventions are used in the directive descriptions. Anything enclosed between '<' and '>' is required variable statement content. Anything enclosed between '[' and ']' is optional.

Anytime the optional `[label]` field is identified in a statement's description it means an optional label, if provided, will be assigned the current statement's address within the active control or dummy section. Any other action for the label field content will be explicitly identified.

Anything in `UPPER CASE` is required as specified in the description.

ASMA Specific Behavior: The following directives are unique to ASMA and are not found in other assemblers:

> `PSW`, `PSWS`, `PSW360`, `PSW67`, `PSWBC`, `PSWEC`, `PSW380`, `PSWXA`, `PSWE370`, `PSWE390`, `PSWZ`, `REGION`, `XMODE`.

The following directives diverge in some respect from legacy usage. See their respective descriptions for details.

> `CCW`, `DC`, `DS`, `END`, `ENTRY`, `START`.

ASMA Limitation: Any assembler directive not described in this manual is not supported by ASMA.

CCW

```
[label] CCW   <command>,<address>,<flags>,<count>
```

The `CCW` directive constructs a Channel Command Word based upon the current execution mode format setting for a CCW. If the CCW execution mode has been set to '`none`', the `CCW` directive is not recognized. See the `XMODE` directive.

ASMA Specific Behavior: The legacy operation of the CCW directive always created a Format-0 CCW for backward compatibility with source predating the existence of the Format-1 CCW. The above description of the CCW directive diverges from this legacy behavior.

The following priority exists for channel command word format generation:

1. An explicit `CCW0` or `CCW1` directive in the program source. An explicit `CCW0` or `CCW1` directive ignores the current `XMODE CCW` setting.
2. The current setting by an explicit `XMODE CCW` directive at the time a `CCW` directive is encountered. An explicit `XMODE CCW` directive overrides the `--ccw` command line argument.
3. A value supplied by the ASMA command-line option `--ccw` establishing the `XMODE CCW` setting preceding an explicit `XMODE CCW` directive. A command line argument overrides the MSL file.
4. The value identified by the MSL file selected by means of the `--target` or `--cpu` ASMA command line argument.

The current setting is made available to a macro by means of the system variable symbol `&SYSCCW`.

CCW0

```
[label] CCW0  <command>,<address>,<flags>,<count>
```

The `CCW0` directive generate a Format-0 Channel Command Word. The channel command word will be double word aligned. All operands are expressions. The `<command>`, `<flags>`, and `<count>` operands must evaluate to an integer. The `<address>` operand must evaluate to an address. All operands are required. All reserved bits are set to zero. Use `DC` directives to build an invalid Format-0 Channel Command Word for testing purposes.

CCW1

```
[label] CCW1  <command>,<address>,<flags>,<count>
```

The `CCW1` directive generates a Format-1 Channel Command Word. All of the operand rules are the same as for the `CCW1` directive. All reserved bits are set to zero. Use `DC` directives to build an invalid Format-1 Channel Command Word for testing purposes.

COPY

```
[label] COPY  'filename'
```

The `COPY` directive inserts into the input stream statements from the file identified between the single quoted string that is the single required argument of the `COPY` directive. Single quotation marks are prohibited within the file name. The file name itself must be compatible with the platform in use. The file is searched within the directories specified by the ASMPATH environment variable and the current active directory.

If the optional `[label]` is supplied, it is ignored.

CSECT

```
[label]  CSECT [comments]
```

The `CSECT` directive initiates a new control section or continues a previously initiated control section. A new control section is added to the current active region. Control sections are double word aligned within the region. The `label` is the symbol assigned to the control section. The symbol's value is the address of the start of the control section. Its length attribute is the total length of the control section. Its image attribute is the position of the control section within the binary image relative to the image's start.

If the optional `label` field is omitted, an unnamed control section is created or continued. Being unnamed, attribute values are not available for an unnamed control section.

If no current active region exists, an unnamed region with a starting address of 0 is created to which the new control section is added.

When a control section is continued, the region within which the control section is defined automatically becomes the current active region.

Any operand data that may be supplied is treated as a comment.

DC

```
[label] DC    [d]A[Ln](address[,...])[,...]
[label] DC    [d]t[Ln]'content[,...]'[,...]
```

The DC directive defines storage usage and content. Usage is specified by a usage

description. Usage descriptions are identical to those used in the DS directive. The content is provided within a pair of single quotes or a left/right parenthesis pair depending upon the type of storage usage. The 'A' and 'AD' types require a parenthesis pair. All other types use the pair of single quotes. Multiple content values may be used where they share storage usage. Multiple definitions may be supplied, separated by a single comma. Multiple values may be supplied for the content except as noted below.

The C, CA and CE constant types allow only a single value to be specified. Two single quotes or ampersands in succession are converted into one single quote or single ampersand within the assembled constant.

The optional duplication [d] factor may be either an unsigned decimal self-defining term or an expression enclosed in parenthesis. The expression must evaluate to an integer of zero or greater.

The optional length modifier [Ln] may be either an unsigned decimal self-defining term or an expression enclosed in parenthesis. The value must be 1, not exceed the maximum allowed for the constant type and not excluded for the type. The initial required L may be either upper or lower case.

This table describes various attributes of the usage definition an content specification. “Signed” implies that a numeric value is preceded by either a plus sign, +, or minus sign, -. If either sign is omitted, a plus sign is assumed. “Unsigned' means that a numeric value is preceded by the letter U.

Type	Implied Length	Implied Alignment	Trunc / Pad	Image Content and Nominal Value
A	4	4	Left	An expression evaluating to an address or integer >=0
AD	8	8	Left	An expression evaluating to an address or integer >=0
B	content	none	Left	Binary data defined by '0' and '1'
C	content	none	Right	EBCDIC character data translated from ASCII input
CA	content	none	Right	ASCII character data as allowed by Python UTF-8 encoding
CE	content	none	Right	EBCDIC character data translated from ASCII input
D	8	8	Left	Signed or unsigned integer (not floating point)
F	4	4	Left	Signed or unsigned integer
FD	8	8	Left	Signed or unsigned integer
H	2	2	Left	Signed or unsigned integer
P	content	none	Left	Signed or unsigned packed decimal data, decimal point optional
S	2	2	none	An expression evaluating to an address in base and 12-bit displacement format
X	content	none	Left	Hexadecimal data defined by hex digits 0-9 and A-F
Y	2	2	Left	An expression evaluating to an address or integer of >= 0
Z	content	none	Right	Signed or unsigned zone decimal data, decimal point optional

If the optional [label] is specified, it acquires the address of the first storage usage description and its length.

ASMA Specific Behavior: The supported constant formats are a subset of those supported by legacy assemblers. Rather than defining a hexadecimal floating point value the D constant type defines an 8-byte signed integer value. It acts as a synonym for the FD constant type. This allows use of the D constant type in legacy programs for storage allocation and large integer definitions where the FD constant type was unavailable.

ASMA Specific Behavior: The plus and minus signs are always treated as expression operators. If used within a duplication factor or length modifier, the expression must be enclosed in parenthesis. For example, a duplication factor of +3 must be coded as (+3).

ASMA Limitations: The following constant types are not supported by ASMA: CU, D, DB, DD, DH, E, EB, ED, EH, G, J, L, LB, LD, LH, LQ, Q, R, RD, SY and V.

ASMA Limitations: Only the length modifier (L) is supported. The scale (S) and exponent (E) modifiers are not supported. The program type sub-field (P) is not supported.

ASMA Limitations: Use of floating point numeric values in constants is not supported.

DROP

```
[label] DROP  <reg>[,reg...]
```

The DROP directive ensures no USING assignment exists for a register. At least one register must be identified. Up to 15 more optional registers may be specified. Each <reg> operand is an expression that must evaluates to an integer between 0 and 15, inclusive.

DS

```
[label]  DS  [d]t[Ln][,...]
```

The DS directive defines storage usage. Usage is specified by a description operand. A description operand includes an optional duplication factor, [d], a required type, t, and an optional explicit length, [Ln]. It an explicit length is specified, any implied alignment of the type is suppressed. Multiple storage usage operands are allowed.

If the optional [label] is specified it acquires the address of the first storage usage description and the length of the storage usage definitions taken as a whole.

See the DC directive description of supported types and implied lengths and alignment.

Within the created image data, areas defined by the DS directive will initialize each storage usage to binary zeros. Other content can replace the binary zero content by use of the ORG directive to overlay the default content.

DSECT

```
<label> DSECT [comments]
```

The DSECT directive initiates or continues a previously initiated dummy section. The label field identifies the symbol associated with the dummy section. The value will always be a section relative address of zero. The label's length will be the total length of the dummy section.

Any operand information provided is treated as a comment.

EJECT

```
[label] EJECT [comments]
```

The EJECT directive introduces a new page in the assembly listing. If the optional [label] is provided, it is ignored. Any operand field information is treated as comments.

END

```
[label] END   [entry]
```

The END directive terminates the assembly. Once encountered, no more input statements are allowed. The optional [label] field will assign to the label a value corresponding to the current address within the active CSECT or DSECT.

The optional [entry] operand defines the entry address of the image.

ASMA Specific Behavior: The ENTRY directive is an alternative method for defining the image entry point. See the ENTRY directive.

ENTRY

```
[label] ENTRY <address>
```

The ENTRY directive defines the entry address of the program within the image. Because the output image file has no inherent structure, no mechanism exists for the communication of the entry point to any actual user of the image file contents. It is reported for use in whatever downstream process would require this information.

The <address> operand is required and may be an expression. The value to which the expression evaluates is reported as the entry point.

An ENTRY directive supersedes any previous ENTRY directive within the assembly. Only the last ENTRY directive is used.

ASMA Specific Behavior: The usage of the ENTRY directive to define a program entry point instead of an entry from an externally linked program diverges from legacy behavior.

EQU

```
<label> EQU   <expression>[,length]
```

The EQU directive assigns a value to the label. The value may be an address or an integer depending on the evaluation of the required expression.

An optional operand, length, may be supplied. The value of the length operand expression specifies the length attribute of the symbol defined by the EQU directive. The length operand .must evaluate to an integer. If omitted, the length attribute of the symbol defaults to one.

MACRO

```
[label] MACRO [DEBUG] [comment]
```

Causes the assembler to exit open code assembly and enter macro definition mode. Any optional label is ignored. If the optional DEBUG operand is provided, macro definition debug output will be generated. Otherwise, any text provided in the operand field is treated as a comment. The macro directive MEND causes the assembler to leave macro definition mode and return to open code assembly. See the “Macro Language” section for details on usage

of the MACRO and MEND directives.

MHELP

```
[label] MHELP <action> [comment]
```

Establishes macro diagnostic information. The required <action> operand is an arithmetic expression composed of binary, decimal or hexadecimal self-defining terms. If an optional [label] is supplied, it is silently ignored. The <action> operand identifies a set of macro language trace, dump and call controls to be enabled until the next MHELP directive is encountered.

Values above 255 control the maximum &SYSNDX value allowed during the assembly. If none of the bits in the next to the last byte of the value are set, &SYSNDX value monitoring is suppressed.

Other trace and dump options and their respective values are described in this table. Only SET symbols defined without subscripts are dumped.

Binary	Decimal	Description
B'00000001'	1	Call trace of macro name, nesting depth and &SYSNDX
B'00000010'	2	Branch trace for AGO and AIF macro directives
B'00000100'	4	Dump SET symbols before AIF executed
B'00001000'	8	Dump SET symbols when MEND or MEXIT encountered
B'00010000'	16	Dump macro parameter values upon macro entry
B'00100000'	32	Suppress dumping global SET symbols with action 4 or 8
B'01000000'	64	Include hexadecimal values of SETC symbols with actions 4, 8 or 16. See below.
B'10000000'	128	Suppresses currently active MHELP actions.

ASMA Specific Behavior: The <action> operand is not limited to a single decimal or binary self-defining term. Macro trace and dump information is sent to the ASMA system output file, not the assembler listing. Redirect the system output to a file if excessive information is generated.

ASMA Specific Behavior: SETC variable symbols support both ASCII and EBCDIC values. When action 64 is used, the EBCDIC hexadecimal values are included in the dump along with the ASCII characters. Depending upon the context, the originating ASCII or translated

EBCDIC value will be used. See the "SETC" macro language section for details.

MNOTE

```
[label] MNOTE <severity>,<message>
```

The MNOTE directive generates an in-line message. The <severity> operand is required. It may be either a numeric value or an asterisk, '*'. The message operand is required and is coded within quotes. The optional [label] field, if provided, is ignored.

ASMA Limitation: The severity operand is supported for compatibility. ASMA does not support severity processing. A severity of '*' **is** treated as an informational message.

ORG

```
[label] ORG   <expression>
```

The ORG directive assigns a new value to the current location counter, represented by '*' within expressions. The required <expression> operand must evaluate to an address within the current active control section.

If the optional label is provided it is assigned the value of the current location counter before the new location is assigned to it.

PRINT

```
[label] PRINT <option>[,option]...
```

The PRINT directive provide listing control settings. Each operand specifies a case insensitive option. At least one option is required. The following options are supported:

- ON – Enables generation of assembler statements in the listing.
- OFF – Disables assembler statements in the listing.
- GEN – Enables macro generated statements in the listing.
- NOGEN – Disables macro generated statements in the listing.
- DATA – Causes all object code generated by a statement in the listing.

- NODATA – Causes a maximum of eight bytes of object code generated by a statement in the listing.

At the start of the assembly the following options are in effect: ON, GEN, and NODATA.

If a label is supplied, it is ignored.

PSW

```
[label] PSW   <sys>,<key>,<a|amwp|mwp>,<prog>,<addr>[,amode]
```

The PSW directive constructs a Program Status Word based upon the current execution mode format setting for a PSW. If the PSW execution mode has been set to 'none', the PSW directive is not recognized. See the XMODE directive.

The following priority exists for Program Status Word format generation:

1. An explicit PSWxx directive in the program source. An explicit PSWxx directive ignores the current XMODE PSW setting.
2. The current setting by an explicit XMODE PSW directive at the time a PSW directive is encountered. An explicit XMODE PSW directive overrides the --psw command line argument.
3. A value supplied by the ASMA command-line option --psw establishing the XMODE PSW setting preceding an explicit XMODE PSW directive. A command line argument overrides the MSL file.
4. The value identified by the MSL file selected by means of the --target or --cpu ASMA command line argument.

The current setting is made available to a macro by means of the system variable symbol &SYSPSW.

PSWS

```
[label]  PSWS  <sys>,<key>,<a>,<prog>,<addr>[,amode]
```

The PSWS directive defines a 32-bit short Program Status Word used on S/360 Model 20. The PSW is not aligned because no alignment requirement exists for S/360 Model 20 PSW's. The label is optional. Five operands are required. Each operand is an expression. The following table describes the operands.

Operand	Identification	PSW Bits	Description
1	<sys>	7	Channel mask
2	<key>	Not Used	Note 1.
3	<a>	6	ASCII Mode bit
4	<prog>	2, 3	Condition Code
5	<addr>	16 - 31	Instruction address or integer
6	[amode]	Not used	Note 2.

Note 1: The `key` operand is not used by `PSWS`. It must be syntactically correct, but is otherwise ignored. It is supported for compatibility with other PSW related directives.

Note 2: The `amode` operand is not used by `PSWS`. If specified it must be syntactically correct but is otherwise ignored. It is supported for compatibility with other PSW related directives.

All reserved bits are set to zero and any required bit settings are enforced. Use `DC` directives to build an invalid PSW for testing purposes.

PSW360

```
[label]  PSW360  <sys>,<key>,<amwp>,<prog>,<addr>[,amode]
```

The `PSW360` directive defines a 64-bit Program Status Word used on all S/360 models other than model 20. The PSW is double word aligned. The label is optional. Five operands are required and the last operand, `amode`, is optional. Each operand is an expression. The following table describes the operands.

Operand	Identification	PSW Bits	Description
1	<sys>	0 - 7	Interrupt masks
2	<key>	8 - 11	Storage protection key
3	<amwp>	12 -15	Note 1.
4	<prog>	34 -39	Condition Code and program mask
5	<addr>	40 - 63	Instruction address
6	<amode>	Not used	Note 2.

Note 1: This operand sets the following PSW bits:

- bit 12 – ASCII mode bit,
- bit 13 – Machine-check interrupt mask
- bit 14 – Wait state
- bit 15 – Program state

Note 2: The `amode` operand is not used by `PSW360`. If specified it must be syntactically correct but is otherwise ignored. It is supported for compatibility with other PSW related directives.

All reserved bits are set to zero and any required bit settings are enforced. Use `DC` directives to build an invalid PSW for testing purposes.

PSW67

```
[label]  PSW67  <sys>,<key>,<amwp>,<prog>,<addr>[,amode]
```

The PSW67 directive creates a S/360 model 67 67-mode PSW. The PSW is double word aligned. The label is optional. Five operands are required and the last operand, `amode`, is optional. The last operand, `amode`, is optional. Each operand is an expression. The following table describes the operands.

Operand	Identification	PSW Bits	Description
1	`<sys>`	5 - 7	Interrupt masks and controls
2	`<key>`	8 - 11	Storage protection key
3	`<amwp>`	12 -15	Note 1.
4	`<prog>`	16 -23	Condition Code and program mask
5	`<addr>`	32 - 63	Instruction address
6	`[amode]`	4	Note 2.

Note 1: This operand sets the following PSW bits:

- bit 12 – ASCII mode bit,
- bit 13 – Machine-check interrupt mask
- bit 14 – Wait state
- bit 15 – Program state

Note 2: This operand defines the address mode. As with the other operands it is an

expression. If omitted, amode defaults to 0, 24-bit address mode. The following values are accepted to specify the address mode:

- 0 – 24-bit address mode,
- 1 – 32-bit address mode,
- 24 – 24-bit address mode, or
- 32 – 32-bit address mode.

All reserved bits are set to zero and any required bit settings are enforced. Use DC directives to build an invalid PSW for testing purposes.

PSWBC

```
[label]  PSWBC <sys>,<key>,<mwp>,<prog>,<addr>[,amode]
```

The PSWBC directive defines a 64-bit S/370 Basic Control Program Status Word. The PSW is double word aligned. The label is optional. Five operands are required and the last operand, amode, is optional. Each operand is an expression. The following table describes the operands. The format bit, bit 12 is set to zero.

Operand	Identification	PSW Bits	Description
1	<sys>	0 - 7	Interrupt masks
2	<key>	8 - 11	Storage protection key
3	<mwp>	13 -15	Note 1.
4	<prog>	34 -39	Condition Code and program mask
5	<addr>	40 - 63	Instruction address
6	[amode]	Not used	Note 2.

Note 1: This operand sets the following PSW bits:

- bit 13 – Machine-check interrupt mask
- bit 14 – Wait state
- bit 15 – Program state

Note 2: The amode operand is not used by PSWBC. If specified it must be syntactically correct but is otherwise ignored. It is supported for compatibility with other PSW related directives.

All reserved bits are set to zero and any required bit settings are enforced. Use DC directives

to build an invalid PSW for testing purposes.

PSWEC

```
[label]  PSWEC <sys>,<key>,<mwp>,<prog>,<addr>[,amode]
```

The PSWEC directive defines a 64-bit S/370 Extended Control Mode Program Status Word. The PSW is double word aligned. The label is optional. Five operands are required and the last operand, amode, is optional. Each operand is an expression. The following table describes the operands. Not all bit settings are allowed on all processors or models. The format bit, bit 12 is set to one.

Operand	Identification	PSW Bits	Description
1	<sys>	0 - 7	Interrupt masks
2	<key>	8 - 11	Storage protection key
3	<mwp>	13 -15	Note 1.
4	<prog>	16 -23	Address space, condition code and program mask
5	<addr>	40 - 63	Instruction address
6	[amode]	Not used	Note 2.

Note 1: This operand sets the following PSW bits:

- bit 13 – Machine-check interrupt mask
- bit 14 – Wait state
- bit 15 – Program state

Note 2: The amode operand is not used by PSWEC. If specified it must be syntactically correct but is otherwise ignored. It is supported for compatibility with other PSW related directives.

All reserved bits are set to zero and any required bit settings are enforced. Use DC directives to build an invalid PSW for testing purposes.

PSW380, PSWXA, PSWE370, PSWE390

```
[label]  PSW380  <sys>,<key>,<mwp>,<prog>,<addr>[,<amode>]
[label]  PSWXA   <sys>,<key>,<mwp>,<prog>,<addr>[,<amode>]
```

```
[label]  PSWE370 <sys>,<key>,<mwp>,<prog>,<addr>[,<amode>]
[label]  PSWE390 <sys>,<key>,<mwp>,<prog>,<addr>[,<amode>]
```

The PSW380, PSWXA, PSWE370 and PSWE390 directives define a 64-bit bimodal address mode Program Status Word used in Hercules specific S/380, 370-XA, ESA/370 and ESA/390 modes, respectively. z/Architecture ® models also support PSWE390 PSW's. The PSW is double word aligned. The label is optional. Five operands are required and the last operand, amode, is optional. Each operand is an expression. The following table describes the operands.

Operand	Identification	PSW Bits	Description
1	<sys>	0 - 7	Interrupt masks
2	<key>	8 - 11	Storage protection key
3	<mwp>	13 -15	Note 1.
4	<prog>	16 -23	Address space, condition code and program mask
5	<addr>	40 -63	Instruction address
6	[amode]	32	Note 2.

Note 1: This operand sets the following PSW bits:

- bit 13 – Machine-check interrupt mask
- bit 14 – Wait state
- bit 15 – Program state

Note 2: This operand defines the address mode. As with the other operands it is an expression. If omitted, amode defaults to 0, 24-bit address mode. The following values are accepted to specify the address mode:

- 0 – 24-bit address mode
- 1 – 31-bit address mode
- 24 – 24-bit address mode
- 31 – 31-bit address mode

All reserved bits are set to zero and any required bit settings are enforced. Use DC directives to build an invalid PSW for testing purposes.

PSWZ

```
[label]  PSWZ  <sys>,<key>,<mwp>,<prog>,<addr>[,<amode>]
```

The `PSWZ` directive defines a 128-bit z/Architecture Program Status Word used on all z/Architecture models. The PSW is double word aligned. The label is optional. Six operands are required. Each operand is an expression. The following table describes the operands.

Operand	Identification	PSW Bits	Description
1	`<sys>`	0 - 7	Interrupt masks
2	`<key>`	8 - 11	Storage protection key
3	`<mwp>`	13 -15	Note 1.
4	`<prog>`	16 - 24	Address space, condition code and program mask
5	`<addr>`	64 -127	Instruction address
6	`[amode]`	31, 32	Note 2.

Note 1: This operand sets the following PSW bits:

- bit 13 – Machine-check interrupt mask
- bit 14 – Wait state
- bit 15 – Program state

Note 2: This operand defines the address mode. As with the other operands it is an expression. If omitted, `amode` defaults to 0, 24-bit address mode. The following expression values are accepted to specify the address mode:

- 0 – 24-bit address mode
- 1 – 31-bit address mode
- 3 – 64-bit address mode
- 24 – 24-bit address mode
- 31 – 31-bit address mode
- 64 – 64-bit address mode.

All reserved bits are set to zero and any required bit settings are enforced. Use `DC` directives to build an invalid PSW for testing purposes.

REGION

```
[label] REGION [comments]
```

The REGION directive continues a previously initiated region. The label identifies the region that is being continued. The control section active in the region at the time it was interrupted automatically becomes the active control section following activation of the region.

If the label is omitted, the previously created unnamed region becomes the active region.

Initiation of a new region requires use of the START directive.

If any operand information is provided it is treated as a comment.

SPACE

```
[label] SPACE  [expression]
```

The SPACE directive inserts one or more spaces into the listing as specified by the expression operand. The expression must consist of binary, decimal or hexadecimal self-defining terms. If the expression operand is omitted, the directive defaults to one space. If the value of the expression exceeds the number of lines per page, an EJECT directive is implied. If supplied, the optional label is ignored.

START

```
[label]  START  [expression][,[region]]
```

The START directive initiates a new control section and optionally a new region. The results of the ASMA START directive model the effects of the first START directive of legacy programs within the context of address regions within an ASMA program. The optional label field identifies the name of the new control section. If the label field is omitted an unnamed control section is created.

The operands to the START directive define the optional new region being created into which the new control section is placed. The expression operand defines the starting address of the region. The region operand identifies the new region name. If the region operand is

present but the `expression` operand is omitted, the starting address of the new region is 0. If the `expression` operand is present but the `region` operand is omitted, a new unnamed region will be created.

If both of the `expression` and `region` operands are omitted, the new control section is placed into the current active region. If no active region exists, an unnamed region is created starting at address 0.

An error condition exists when the named or unnamed region or control section that would be created already exists.

For a `START` directive to which comments are applied but neither operand is used, a comma is required in the operand field. Otherwise the the comments will be interpreted as the start of the `expression` operand.

The following table summarizes the possible cases and the actions taken by the `START` directive. The address defined by the `<expression>` operand applies to the created region. The address assigned to the control section is dictated by the placement within the region to which it is added. If it is the first control section of the region, it will have the same assigned address as the region. If it is added to a region already containing one or more control sections, the control section will start on the next doubleword following the preceding control section in the region.

<label>	<expression>	<region>	Action
omitted	omitted	omitted	Unnamed control section created in the current active region. If no active region, the unnamed region is created starting at address 0.
omitted	present	omitted	Unnamed control section created in a new unnamed region starting at the address of the expression
omitted	omitted	present	Unnamed control section created in a new named region starting at address 0.
omitted	present	present	Unnamed control section created in a new named region starting at the address of the expression
present	omitted	omitted	Named control section created in the current active region. If no active region, the unnamed region is created starting at address 0.
present	present	omitted	Named control section created in a new unnamed region starting at the address of the expression.
present	omitted	present	Named control section created in a new named region starting at address 0.
present	present	present	Named control section created in a new named region starting at the address of the expression.

ASMA Specific Behavior: ASMA supports multiple regions, each with its own independent location counter. Only the `START` directive creates a new region. Multiple `START` directives are allowed to support multiple regions. For programs written expressly for use of ASMA, it is recommend that all operands and the label field be used to minimize interplay of named and unnamed regions and sections.

ASMA Specific Behavior: The unnamed region will be assigned a file name of `unnamed.bin` when the `--gldipl` option is used. All other regions utilize their assigned name when generating list directed IPL output.

TITLE

```
[label] TITLE  'New Listing Title'
```

The `TITLE` directive allows specification of a new title for the assembly listing. The title may contain upper or lower case letters and any other character other than a single quote. Two adjacent single quotes or ampersands are required to represent a single quote or single ampersand, respectively, in the printed listing title.

If the optional `[label]` is provided, it is ignored.

USING

```
[label] USING <address>,<reg>[,<reg>]
```

The `USING` directive establishes one or more base registers for the resolution of storage accesses into a base and displacement within machine instructions. The `<address>` and each supplied `<reg>` operand are expressions. The `<address>` expression must evaluate to an address or location within a dummy section. A `<reg>` expression must evaluate to an integer between 0 and 15 inclusive.

Each additional register is assigned a base value 4096 bytes from the previous assigned address.

A base register assignment in place for any of the specified registers replaces a previous assignment without the need of a `DROP` directive.

ASMA Specific Behavior: Because `USING` directive processing occurs during Pass 2 statement processing following the assignment of absolute address locations, a `USING` directive associated with a control section will result in one or more registers being associate with the absolute address. Any address validly in range for the instruction's displacement

field is accepted regardless of control sections involved in the USING directive address expression, the instruction itself or an instruction's operand. Instruction operands targeting dummy section labels require the same dummy section to be associated with the `USING` address expression as occurs in the instruction operand.

XMODE

```
[label] XMODE CCW,<0|1|none>
[label] XMODE PSW,<S|360|67|BC|EC|XA|E370|E390|Z|none>
```

The `XMODE` directive sets the current CCW or PSW execution mode. In both cases the execution mode determines the specific PSW format or CCW format created by the `PSW` or `CCW` directive, respectively. In either case, specifying `none` removes the existing setting and causes the `PSW` or `CCW` directive to not be recognized by the assembler.

The first occurrence of the `XMODE` directive will override a command line value provided by either the `--ccw` or `--psw` option. If a command line option is not specified, the default is the expected CCW or PSW format defined for the CPU in the MSL database.

If the optional `[label]` is supplied, it is ignored.

Macro Language

ASMA supports two separate languages:

- the ASMA assembler language, described previously, is primarily declarative in nature, and
- the ASMA macro language, described in this section, is primarily interpretive and results in the creation of declarative assembler language statements.

The ASMA macro language processing provides a subset of some legacy implementations and a super set of others. It should not be expected that any legacy macro definition will work without some changes to accommodate the ASMA macro language.

The macro language recognizes two types of statements. Statements that occur in open code, outside of a macro definition, and statements that occur within a macro definition. Macro Language directives may only occur within a macro definition.

ASMA Limitation: The concept of macro libraries does not apply. All user defined macros must be present in the source statements or included by means of the `COPY` assembler directive. Use the `PRINT OFF` and `PRINT ON` assembler directives to eliminate the

definitions from the listing if desired.

ASMA Limitation: Nested macro definitions are not supported. A macro definition occurring within another macro definition will result in as many as six forms of errors.

1. The inner macro definition's `MACRO` statement will generate an error and will be ignored.
2. The inner macro definition's prototype statement will be treated as a model statement within the outer definition.
3. The inner macro's body statements will be erroneously treated as additional body statements of the outer macro definition.
4. The `MEND` associated with the inner macro definition will result in a premature end of the outer macro definition being interpreted as the `MEND` of the outer definition.
5. The outer macro's body statements following the MEND associated with the inner macro definition will be treated as occurring in open code.
6. The `MEND` associated with the outer macro definition, when finally encountered in open code, will generate an error.

Do not use inner macro definitions!

Macro Definition Mode

Macro definition mode is entered when the assembler directive `MACRO` is recognized during open code assembly. A user macro is defined within the constraints of the `MACRO` assembler directive and a following by a `MEND` macro directive. Subsequent definitions of a macro with the same name supersede previous definitions.

The first non-comment statement following the `MACRO` assembler directive must be the prototype statement.

Following the prototype statement is the optional macro body. The macro body consists of

- macro directives, described below, and
- model statements.

Macro directives are processed interpretively during the invocation of the macro. A macro is invoked when its name occurs in the operation field of an open code or macro model statement.

ASMA Specific Behavior: Errors occurring in macro directives or a macro's prototype statement will cause the macro definition to be ignored. Attempts to invoke a failed macro definition will result in an unrecognized statement.

Built-In Macros

Macros provided by ASMA without requiring user definitions are called “built-in” macros. Built-

in macros follow the same usage rules as do other macro definitions. However, they are directly implemented in Python directly utilizing the internal representation of any macro definition. Built-in macros may also extend the operations available beyond those available to user defined macros.

Presently only a single built-in macro is defined for the purposes of testing support for built-in macros, namely, the `TEST` macro. It may be redefined by a user defined macro.

Macro Invocation

A macro definition is invoked when it occurs in open code assembly. The invocation creates a new input source. The statements of the macro definition in their internal form are interpreted. When a model statement is encountered it has any of its embedded symbolic parameters replaced and is then submitted back to the assembler as an open code statement from the macro input source.

ASMA Specific Behavior: Error handling during macro invocation uses a “fail-on-error” strategy. An error encountered during invocation causes immediate termination of the macro. The error is reported with the invoking open code statement. Included in the error is the assembly statement of the failing macro definition.

Macro Symbols

Two types of symbols are unique to the macro language:

1. the variable symbol – any normal label name preceded by an ampersand, '`&`', and
2. the sequence symbol – any normal label name preceded by a period, '`.`'.

Sequence symbols may occur only in the name field of a macro definition's body. They are ignored in open code. The sequence symbol identifies where control is to be passed during the invocation of the macro. Sequence symbols are local to the macro.

Variable symbols arise in three contexts:

- prototype parameters, either as positional or keyword defined parameters
- global variable symbols or
- local variable symbols, and
- system variable symbols.

Within the context of an invoked macro, a single name space is used for all variable symbols. Symbol names occurring as parameters, globally declared or locally declared symbols may must be unique.

Parameter variable symbols are “read-only” and are assigned values when the macro is invoked. All parameter variable symbols contain characters as if they were declared as local character variable symbols.

Global variable symbols are shared among macros. The first occurrence of the symbol's declaration creates it, after which it is shared between open code and invoked macros.

Local variable symbols are declared within the invoking macro and are only usable within the invoked macro. Undeclared local variable symbols are not supported.

System variable symbols will have a local or global context and type depending upon the symbol. System variable symbols all start with '`&SYS`'. User defined macros should not use these characters at the start of user defined variable symbols. Normal global and local symbol rules apply.

Variable symbols are of three types:

- arithmetic, assigned a signed numeric value, see the `LCLA` and `GBLA` macro directives, or
- binary, assigned a value of either zero or one, see the `LCLB` and `GBLB` macro directives, or
- character, assigned a string value, see the `LCLC` and `GBLC` macro directives.

In the following discussion, descriptions will use:

- `label` – for normal assembler location symbols,
- `var` – for a variable symbol and
- `seq` – for sequence symbols.

Subscripts

Globally or locally defined variable symbols may be defined with subscripts. Subscripts require the use of a self defining value or an arithmetic variable symbol to identify the subscript. Variable symbol subscripts must be in the range of one to the number of subscripts defined for the symbol.

ASMA Specific Behavior: Subscript zero is equivalent to the unsubscripted variable. All variable symbols are in essence defined with subscripts. The default subscript is zero. For example, accessing symbol `&sym` is the same as accessing `&sym(0)` explicitly.

Variable Symbol Attributes

Two attributes are supported for macro symbolic variables:

- count (`K'`) – the count of characters used to represent the symbolic variable or referenced subscript.
- number (`N'`) – the number of subscripts defined for the symbolic variable or zero if not defined with subscripts.

Both attributes result in a arithmetic value that may be used anywhere a numeric value is

valid.

The count attribute always results in 0 for arithmetic or binary variable symbols regardless of the symbolic variable or subscripts actual value.

The number attribute is supported for all variable symbols regardless of their defined type. Subscripts are never allowed when referencing the number attribute.

System Variable Symbols

System variable symbols are read-only global or local symbols made available to a macro as if the corresponding global or local declaration had been made by the macro. The values established at the time a macro is invoked remain constant for the duration of that specific macro. The following system variable symbols are supported.

Symbol	Type	Description
&SYSASM	GBLC	Name of the assembler: 'A SMALL MAINFRAME ASSEMBLER'
&SYSCCW	LBLC	'CCWn' directive used by the CCW directive, or '' if not set
&SYSCLOCK	LBLC	Assembly UTC time in 'YYYY-MM-DD HH:MM:SS.mmmmmm' format
&SYSDATC	GBLC	Assembly local date in 'YYYYMMDD' format.
&SYSDATE	GBLC	Assembly local date in 'MM/DD/YY' format
&SYSECT	LCLC	Name of the current active CSECT
&SYSMAC	LBLC	Macro name
&SYSNDX	LBLC	Macro index number
&SYSNEST	LBLA	Macro nesting level
&SYSPSW	LBLC	'PSWxxx' used by the PSW directive, or '' if not set
&SYSTIME	GBLC	Assembly local time in 'HH.MM' format
&SYSVER	GBLC	ASMA version in 'V.R.M' format

Prototype Statement

```
[var]  <macname> [<var>[=<default>]][,<var>[=<default>]]...
```

Macro Definition Mode

The first statement in a macro definition following the MACRO assembler directive is the prototype statement.

The name field may contain an optional variable symbol.

The operation field contains the name of the macro. It follows the same rules as those of normal symbols. It is the name by which the assembler recognizes the macro when invoked during open code assembly.

The operand field contains a list of comma separated parameters. A positional parameter is a single variable symbol. A keyword parameter is a variable symbol, followed immediately by an '=' sign, followed immediately by a character sequence defining the keyword's default value.

Positional and keyword parameters may be mixed within the prototype statement.

Macro Invocation

If provided, the variable symbol in the name field will be assigned the character string value of the normal symbol occurring in the invoking statement's name field.

When the macro is invoked the positional parameter is assigned the string supplied for the corresponding positional parameter in the invoking open code statement. When the macro is invoked keyword parameters take on the corresponding value assigned to the symbol by the invoking open code statement or its default value as defined in the prototype. Positional and keyword parameters are identified by the absence or presence of a equal sign, '=', in the operand. Keyword parameters when invoked do not require the preceding ampersand for the variable symbol whose value is being specified.

Positional parameters are assigned values as they occur in the invoking statement.

Model Statements

All statements within a macro definition that are neither a prototype statement nor a macro directive are model statements. Model statements are scanned for the occurrence of a variable symbol whose string value replaces the occurrence of the variable symbol. An optional period, '**.**', may be used to separate a variable symbol's name from any following statement characters that may follow.

If the variable symbol is defined with subscripts, a subscript must immediately follow the variable symbol in the model statement. The subscript itself may refer to a defined SETA symbol.

Undefined variable symbols occurring in the the model statement are not replaced. Depending upon where the undefined variable symbol occurs, the resultant statement may trigger other assembler errors. Variable symbols occurring in open code are not replaced.

The resulting statement after variable replacement has occurred is submitted to the assembler language for processing and it is this resultant statement that appears in the listing, not the statement before substitution has occurred. Reference to the originating source statement in a macro definition is required to observe the original variable symbols before replacement.

Arithmetic Expressions

Arithmetic expressions support two types of operands:

- self-defining terms, and
- symbolic variable references previously defined by either a `GBLA` or `LCLA` directive, with or without subscripts as dictated by the symbol definition.

Self defining terms are either a

- unsigned decimal number, any combination of number characters, '`0`' through '`9`'; a
- binary value of the form `B'bb...'`, where each '`b`' represents either a '`0`' or '`1`'; a
- hexadecimal value of the form `X'hh...'`, where each '`h`' represent a number '`0`' through '`9`' or upper or lower case '`a`' through '`f`'; or a
- character value of the form `C'e'` or `CA'a'` or `CE'e'`.

Character self defining terms allow for a single character. An EBCDIC character results from the translation of the ASCII source character after conversion by the current code page assembler setting.

An arithmetic expression supports four dyadic operations: addition ('`+`'), subtraction ('`−`'), multiplication ('`*`') and division ('`/`'). Division discards any encountered remainder.

Two monadic operations are supported: positive ('`+`') and negative ('`−`'). A signed self defining term is treated as having a monadic operator.

Parenthesis have the usual effect of overriding operator precedence.

ASMA Limitation: Regular assembler label assignments even to absolute values are not supported by macro language arithmetic expressions.

Logical Expressions

Logical expressions support two types of operands:

- self-defining terms as described in the “Arithmetic Expressions” section;
- symbolic variable references previously defined by either a `GBLA`, `GBLB`, `LCLA` or `LCLB` directive, with or without subscripts as dictated by the symbol definition; or
- character strings, with or without symbolic replacement as occurring for model statements.

A character string is two single quotation marks with zero or more intervening characters, other than a another single quotation mark. Sub-string specifications are not supported in logical expressions.

Arithmetic symbolic variables as operands are treated reflect the value compared to zero. An arithmetic value equal to zero is treated as a binary value of zero, otherwise the value is

considered to be one in the logical expression.

An arithmetic symbolic variable may be compared to a self-defining term or another arithmetic symbolic variable. A character string may be compared to another character string. When character strings are compared, their EBCDIC character coding values are used as established by the current active code page translation. See the “Code Pages” section for details on using built-in code page definitions or providing a user defined code page definition.

The following comparisons are supported. Comparison operators compare value to the left of the operator to the value to the right of the operator. Comparison operates must be in upper case.

- `EQ` – the left hand value equal comparison
- `NE` – not equal comparison,
- `LT` – less than comparison,
- `LE` – less than or equal comparison,
- `GT` – greater than comparison, and
- `GE` – greater than or equal comparison.

Logical expression operands or the results of comparison operators may be combined with logical relationships. Logical relationships combine the operand or comparison operator result on the left with the operand or comparison operator on the right. The following relationships are supported and must be in upper case.

- `AND`,
- `AND NOT`,
- `OR`,
- `OR NOT`,
- `XOR`, and
- `XOR NOT`.

A logical relationship, operand or comparison result may be reversed using the monadic `NOT` operator, in upper case.

The precedence of these operations are from highest to lowest, the highest being performed first if not explicitly controlled via expression parentheses:

- `NOT` monadic operator
- comparison operators
- relationship operators.

The result of a logical expression is either a zero (interpreted as False) or a one (interpreted

as True).

Macro Directives

Macro directives may have roles during either macro definition mode or during macro invocation or both.

AGO

Unconditional Branch Syntax:

```
[seq] AGO <seq>
```

Computed Branch Syntax:

```
[seq] AGO <(arithmetic-expression)><seq>[,seq]...
```

The AGO directive causes either an unconditional branch within a macro to the statement identified by the required sequence symbol (the unconditional branch syntax) or a branch conditional on the results of the required arithmetic expression to one or more locations (the computed branch syntax). The `AGO` directive itself may contain a sequence symbol in its name field. In both cases as least one sequence symbol is required. In the case of the computed branch syntax, additional sequence symbols may be supplied, each separated by a comma.

Macro Definition Mode

In both cases, the `AGO` directive is converted to a internal representation. Sequence symbols are validated as defined within the macro at the completion of the macro definition.

The arithmetic expression in the computed branch syntax must be enclosed within a parenthesis pair and be immediately followed by the initial required sequence symbol. Processing of the arithmetic expression follows that of the `SETA` directive.

Macro Invocation

The unconditional branch syntax form of the directive causes an immediate transfer of control to the macro body statement to which the sequence symbol is associated.

The computed branch syntax form of the directive, first calculates the result of the arithmetic expression. If the result is between one and number of sequence symbols provided, control is transferred to the statement associated with the corresponding sequence symbol. If the result of the arithmetic expression falls outside of this range, control flows to the next sequential statement immediately following the `AGO` directive.

See the “Arithmetic Expressions” section for details related to macro language arithmetic expressions.

AIF

```
[seq] AIF (<logical-expression>)<seq>
```

The AIF directive performs a conditional transfer of control within a macro definition. A required logical expression, enclosed in a parenthesis pair, must be immediately followed by a sequence symbol defined in the name field of a macro directive or model statement.

Macro Definition Mode

The logical expression and sequence symbol are both converted to an internal form for later evaluation. See the “Logical Expressions” sections for details.

Macro Invocation

The logical expression is evaluated. If the expression results in one (implying True), control is passed to the statement that defines the AIF referenced sequence symbol. Otherwise, control flows to the next sequential directive or model statement.

ANOP

```
[seq] ANOP [comment]
```

Macro Definition Mode

The ANOP directive provides a placeholder for defining a sequence symbol, if provided.

Macro Invocation

The ANOP performs no operation during macro invocation.

GBLA

```
[seq] GBLA <sym[(sub)]>[,sym[(sub)]...
```

Macro Definition Mode

The GBLA directive defines one or more global arithmetic symbolic variables.

If the optional subscript is provided for the symbol, the symbol will be defined with subscripts and may only be referenced with a subscript. The subscript must be a decimal self-defining

term.

A sequence symbol in the name field is optional.

Each defined symbol is converted to an internal form used during macro invocation.

Macro Invocation

Each symbol identified in the `GBLA` directive is defined in the global macro symbol table if not already defined and initialized with a value of zero.

If the symbol is already defined its original definition is used.

The global variable is then made available to the macro with its current value if it does not conflict with a prototype parameter or another symbol of the same name already available to the macro.

GBLB

```
[seq] GBLB <sym[(sub)]>[,sym[(sub)]...
```

Macro Definition Mode

The `GBLB` directive defines one or more global binary symbolic variables.

If the optional subscript is provided for the symbol, the symbol will be defined with subscripts and may only be referenced with a subscript. The subscript must be a decimal self-defining term.

A sequence symbol in the name field is optional.

Each defined symbol is converted to an internal form used during macro invocation.

Macro Invocation

Each symbol identified in the `GBLB` directive is defined in the global macro symbol table if not already defined and initialized with a value of zero implying false.

If the symbol is already defined its original definition is used.

The global variable is then made available to the macro with its current value if it does not conflict with a prototype parameter or another symbol of the same name already available to the macro.

GBLC

```
[seq] GBLC <sym[(sub)]>[,sym[(sub)]...
```

Macro Definition Mode

The `GBLC` directive defines one or more global character symbolic variables.

If the optional subscript is provided for the symbol, the symbol will be defined with subscripts and may only be referenced with a subscript. The subscript must be a decimal self-defining term.

A sequence symbol in the name field is optional.

Each defined symbol is converted to an internal form used during macro invocation.

Macro Invocation

Each symbol identified in the `GBLC` directive is defined in the global macro symbol table if not already defined and initialized with a value of the empty character string (`''`).

If the symbol is already defined its original definition is used.

The global variable is then made available to the macro with its current value if it does not conflict with a prototype parameter or another symbol of the same name already available to the macro.

LCLA

```
[seq] LCLA <sym[(sub)]>[,sym[(sub)]...
```

Macro Definition Mode

The `LCLA` directive defines one or more local arithmetic symbolic variables.

If the optional subscript is provided for the symbol, the symbol will be defined with subscripts and may only be referenced with a subscript. The subscript must be a decimal self-defining term.

A sequence symbol in the name field is optional.

Each defined symbol is converted to an internal form used during macro invocation.

Macro Invocation

Each symbol identified in the `LCLA` directive is defined in the local macro symbol table, initialized with a value of zero, and made available to the macro provided the symbol does not conflict with a prototype parameter or another symbol of the same name already made available to the macro.

LCLB

```
[seq] LCLB <sym[(sub)]>[,sym[(sub)]...
```

Macro Definition Mode

The `LCLB` directive defines one or more local binary symbolic variables.

If the optional subscript is provided for the symbol, the symbol will be defined with subscripts and may only be referenced with a subscript. The subscript must be a decimal self-defining term.

A sequence symbol in the name field is optional.

Each defined symbol is converted to an internal form used during macro invocation.

Macro Invocation

Each symbol identified in the `LCLB` directive is defined in the local macro symbol table, initialized with a value of zero, implying false, and made available to the macro provided the symbol does not conflict with a prototype parameter or another symbol of the same name already made available to the macro.

LCLC

```
[seq] LCLC <sym[(sub)]>[,sym[(sub)]...
```

Macro Definition Mode

The `LCLC` directive defines one or more local character symbolic variables.

If the optional subscript is provided for the symbol, the symbol will be defined with subscripts and may only be referenced with a subscript. The subscript must be a decimal self-defining term.

A sequence symbol in the name field is optional.

Each defined symbol is converted to an internal form used during macro invocation.

Macro Invocation

Each symbol identified in the `LCLC` directive is defined in the local macro symbol table, initialized with a value of the empty string (`''`), and made available to the macro provided the symbol does not conflict with a prototype parameter or another symbol of the same name already made available to the macro.

MEND

```
[seq] MEND [comment]
```

Macro Definition Mode

Terminates and completes a macro definition. A completed macro definition contains an internal representation of the definition. `MEND` causes the assembler to leave macro definition mode and return to open code assembly mode. An optional sequence symbol, if provided, is incorporated as part of the defined macro. Any text provided in the operand field is treated as a comment.

If errors were encountered during macro definition mode, the macro definition fails and it is not created. This case is reported as an error occurring during `MEND` processing. Subsequent statements attempting to utilize the failed macro definition will themselves result in unrecognized operations.

Macro Invocation

When interpreted during macro invocation, the `MEND` statement terminates the invocation. Macro interpretation ceases and the macro ceases to be an input source to open code assembly.

MEXIT

```
[seq] MEXIT [comment]
```

Macro Definition Mode

An optional sequence symbol as allowed. Any content of the statement operand field is treated as a comment.

Macro Invocation

Causes the macro to pass control to the macro's `MEND` directive.

SETA

```
<var[(subscript)]> SETA <arithmetic expression>
```

Macro Definition Mode

The name field of the statement must contain a variable symbol which may optionally contain a subscript enclosed in parenthesis. If provided the subscript may be a decimal self defining term a variable symbol previously defined by a `GBLA` or `LCLA` directive. It identifies to what symbol or its subscript the result of the arithmetic expression will be be assigned.

The arithmetic expression is recognized and converted into an internal representation used during macro invocation. Lexical errors are recognized during macro definition mode. Some syntactical errors will not be detected until macro invocation.

Macro Invocation

The arithmetic expression is evaluated. The current value of the subscript (if provided) is determined. The result is assigned to the `GBLA` or `LCLA` variable symbol or its specified subscript.

The presence or absence of a subscript must be consistent with the definition of the symbolic variable or an error is recognized. Symbolic variables defined with a subscript must have a subscript when setting its value. Symbolic variables not defined with a subscript must not have a subscript when setting its value.

Certain syntactical errors may be detected during invocation. Errors related to symbol definitions and subscript values (for example out of range for the symbols definition) are detected during invocation.

See the “Arithmetic Expressions” section for details related to macro language arithmetic expressions.

SETB

```
<var[(subscript)]> SETB <logical expression>
```

Macro Definition Mode

The name field of the statement must contain a variable symbol which may optionally contain a subscript enclosed in parenthesis. If provided the subscript may be a decimal self defining term a variable symbol previously defined by a `GBLA` or `LCLA` directive. It identifies to what symbol or its subscript the result of the arithmetic expression will be be assigned.

The logical expression is recognized and converted into an internal representation used during macro invocation. Lexical errors are recognized during macro definition mode. Some syntactical errors will not be detected until macro invocation.

Macro Invocation

The logical expression is evaluated. The current value of the subscript (if provided) is determined. The result is assigned to the `GBLB` or `LCLB` variable symbol or its specified subscript.

The presence or absence of a subscript must be consistent with the definition of the symbolic variable or an error is recognized. Symbolic variables defined with a subscript must have a subscript when setting its value. Symbolic variables not defined with a subscript must not have a subscript when setting its value.

Certain syntactical errors may be detected during invocation. Errors related to symbol definitions and subscript values (for example out of range for the symbols definition) are detected during invocation.

See the “Logical Expressions” section for details related to macro language arithmetic expressions.

SETC

```
<var[(subscript)]> SETC <'characters'[(start,length)]>
```

Macro Definition Mode

The name field of the statement must contain a variable symbol which may optionally contain a subscript enclosed in parenthesis. If provided the subscript may be a decimal self defining term a variable symbol previously defined by a `GBLA` or `LCLA` directive. It identifies to what symbol or its subscript the result of the character expression will be be assigned.

The character expression is recognized and converted into an internal representation used during macro invocation. Lexical errors are recognized during macro definition mode. Some syntactical errors will not be detected until macro invocation.

Macro Invocation

The character expression is evaluated. The current value of the subscript (if provided) is determined. The result is assigned to the `GBLC` or `LCLC` variable symbol or its specified subscript.

Character expression evaluation occurs in three steps. The second and third occur only if the optional sub-string specification is provided.

1. Perform symbolic variable substitution on the supplied characters between the two single quotation marks. The string may be empty
2. If the sub-string specification is provided, calculate the starting position in the string by evaluating the `start` arithmetic expression and calculate the length of the sub-string by evaluating the `length` arithmetic expression. See the section “Arithmetic Expressions” on details of how the two values are specified.
3. Extract the sub-string from the character sequence.

The result is then assigned to the previously defined GBLC or LCLC symbolic variable. Errors relating to the sub-string or subscript are detected during macro invocation.

Appendix A - Instruction Source Formats

Each mainframe instruction falls into a specific format. All instructions that share the same format also share the same assembler source syntax and binary machine instruction structure. Binary machine instructions identify their components by a letter and number.

Operands in assembler source are separated by commas. All operands are required unless otherwise indicated by a Note. The source syntax or operands are identified in the following table by the heading columns with “Op1” through “Op5”. The operand description includes an alphabetic component and a number. The alphabetic designation indicates the kind of operand and the numbered portion of the machine instruction format.

Assembler syntax operand ordering does not always match instruction operand. For example, in the RIE-d format, the machine instruction identifies the immediate field as its second operand. However, the assembler syntax places the immediate field as its third operand.

The actual formats and instructions supported by ASMA is controlled by the contents of an external database coded using the Machine Specification Language. The command line argument identifying the targeted machine dictates instruction support. The following tables are informational.

Syntax Summary by Instruction Format

Format	Example	Length	Opcode	Op1	Op2	Op3	Op4	Op 5
E bits	PR	2	0-15	-	-	-	-	
I bits	SVC	2	0-7	I1 8-15	-	-	-	-
IE bits	NIAI	4	0-15	I1 24-27	I2 28-31	-	-	-
MII bits	BPRP	6	0-7	M1 8-11	RI2 12-23	RI3 24-47	-	-
RI a bits	IIHL	4	0-7, 12-15	R1 8-11	I2 - Note 1 16-31	-	-	-
RI b bits	BRAS	4	0-7, 12-15	R1 8-11	RI2 16-31	-	-	-
RI c bits	BRC	4	0-7, 12-15	M1 – Note 1 8-11	RI2 16-31	-	-	-
RIE a bits	CGIT	6	0-7 40-47	R1 8-11	I2 16-31	M3 32-35	-	-
RIE b	CGRJ	6	0-7,	R1	R2	M3	RI4	-

Format	Example	Length	Opcode	Op1	Op2	Op3	Op4	Op 5
bits			40-47	8-11	12-15	32-35	16-31	
RIE c bits	CGIJ	6	0-7, 40-47	R1 8-11	I2 32-39	M3 12-15	RI4 16-31	-
RIE d bits	AHIK	6	0-7, 40-47	R1 8-11	R3 12-15	I2 16-31	-	-
RIE e bits	BRXHG	6	0-7, 40-47	R1 8-11	R3 12-15	RI2 16-31	-	-
RIE f bits	RXSBG	6	0-7, 40-47	R1 8-11	R2 12-15	I3 16-23	I4 24-31	I5 32-29
RIL a bits	LGFI	6	0-7. 12-15	R1 8-11	I2 16-47	-	-	-
RIL b bits	LARL	6	0-7 12-15	R1 8-11	RI2 16-47	-	-	-
RIL c bits	BRCL	6	0-7, 12-15	M1 8-11	RI2 16-47	-	-	-
RIS bits	CGIB	6	0-7, 40-47	R1 8-11	I2 32-29	M3 12-15	D4(B4) B4 16-19 D4 20-31	-
RR bits	BCR BALR	2	0-7	M1/R1 8-11	R2 – Note 1 12-15	-	-	-
RRD bits	MAEBR	4	0-15	R1 16-19	R3 24-27	R2 28-31	-	-
RRE bits	LPEBR	4	0-15	R1 - Note 1 24-27	R2 – Note 1 28-31	-	-	-
RRF a bits	MDTR	4	0-15	R1 24-27	R2 28-31	R3 16-19	M4 – Note 1 20-23	-
RRF b bits	QADTR	4	0-15	R1 24-27	R3 16-19	R2 28-31	M4 – Note 1 20-23	-
RRF c bits	CGRT	4	0-15	R1 24-27	R2 28-31	M3 – Note 1	-	-
RRF d bits	LDETR	4	0-15	R1 24-27	R2 28-31	M4 20-23	-	-
RRF e bits	CELFBR	4	0-15	R1 24-27	M3 16-19	R2 28-31	M4 – Note 1 20-23	-
RRS bits	CGRB	6	0-7, 40-47	R1 8-11	R2 12-15	M3 32-35	D4(B4) B4 16-19 D4 20-31	-
RS a bits	BXH	4	0-7	R1 8-11	R3 – Note 1 12-15	D2(B2) B2 16-19 D2 20-31	-	-
RS b bits	STCM	4	0-7	R1 8-11	M3 12-15	D2(B2) B2 16-19	-	-

Format	Example	Length	Opcode	Op1	Op2	Op3	Op4	Op 5
						D2 20-31		
RSI bits	BRXH	4	0-7	R1 8-11	R3 12-15	RI2 16-31	-	-
RSL a bits	TP	6	0-7 40-47	D1(L1,B1) L1 8-11 B1 16-19 D1 20-31	-	-	-	-
RSL b bits	CDZT	6	0-7, 40-47	R1 32-36	D2(L2,B2) L2 8-15 B2 16-15 D2 20-31	M3 36-39	-	-
RSY a bits	LMY	6	0-7, 40-47	R1 8-11	R3 12-15	D2(B2) B2 16-19 DL2 20-31 DH2 32-29	-	-
RSY b bits	CLMH	6	0-7, 40-47	R1 8-11	M3 12-15	D2(B2) B2 16-19 DL2 20-31 DH2 32-40	-	-
RX a bits	STH	4	0-7	R1 8-11	D2(X2,B2) X2 12-15 B2 16-19 D2 20-31	-	-	-
RX b bits	BC	4	0-7	M1 8-11	D2(X2,B2) X2 12-15 B2 16-19 D2 20-31	-	-	-
RXE bits	LDEB	6	0-7, 40-47	R1 8-11	D2(X2,B2) X2 12-11 B2 16-19 D2 20-31	-	-	-
RXF bits	MAEB	6	0-7, 40-47	R1 32-35	R3 8-11	D2(X2,B2) X2 12-11 B2 16-19 D2 20-31	-	-
RXY a bits	LRAG	6	0-7, 40-47	R1 8-11	D2(X2,B2) X2 12-15 B2 16-19 DL2 20-31 DH2 32-39	-	-	-
RXY b bits	PFD	6	0-7, 40-47	M1 8-11	D2(X2,B2) X2 12-15 B2 16-19 DL2 20-31 DH2 32-39	-	-	-
S	STIDP	4		D2(B2)	-	-	-	-

Format	Example	Length	Opcode	Op1	Op2	Op3	Op4	Op 5
bits			0-15	B2 16-19 D2 20-31				
SI bits	TM	4	0-8	D1(B1) B1 16-19 D1 20-31	I2 8-15	-	-	-
SIL bits	MVHHI	6	0-15	D1(B1) B1 16-19 D1 20-31	I2 32-47	-	-	-
SIY bits	TMY	6	0-7, 40-47	D1(B1) B1 16-19 DL1 20-31 DH1 32-39	I2	-	-	-
SMI bits	BPP	6	0-7	M1 9-11	RI2 32-47	D3(B3) B2 16-19 D3 20-31	-	-
SS a bits	TRTR	6	0-7	D1(L,B1) L 8-15 B1 16-19 D1 20-31	D2(B2) B2 32-35 D2 36-47	-	-	-
SS b bits	MVO	6	0-7	D1(L1,B1) L1 8-11 B1 16-19 D1 20-31	D2(L2,B2) L2 12-15 B2 32-35 D2 36-47	-	-	-
SS c bits	SRP	6	0-7	D1(L1,B1) L1 8-11 B1 16-19 D1 20-31	D2(B2) B2 32-35 D2 36-47	I3 12-15	-	-
SS d bits	MVCK	6	0-7	D1(R1,B1) R1 8-11 B1 16-19 D1 20-31	D2(B2) B2 32-35 D2 36-47	R3 12-15	-	-
SS e bits	PLO	6	0-7	R1 8-11	D2(B2) B2 16-19 D2 20-31	R3 12-15	D4(B4) B4 32-35 D4 36-47	
SS f bits	PKA	6	0-7	D1(B1) B1 16-19 D1 20-31	D2(L2,B2) L2 8-15 B2 32-35 D2 36-47	-	-	-
SSE bits	LASP	6	0-15	D1(B1) B1 16-19 D1 20-31	D2(B2) B2 32-35 D2 36-47			
SSF bits	MVCOS	6	0-7, 12-15	D1(B1) B1 16-19 D1 20-31	D2(B2) B2 32-35 D2 36-47	R3 8-11		

Note 1: Not defined for all instructions of this format.

Extended Mnemonics

Extended mnemonics for relative instructions come with two forms, one using BRxxx for BRANCH RELATIVE ON CONDITION modeled on the extended mnemonics for BRANCH ON CONDITION. The other form uses Jxxxx to imply a relative instruction. ASMA will only provide the Jxxxx form of extended mnemonics for relative instructions.

Mnem.	Actual	Implied	Op1	Op2	Op3	Op4	Op5
B	BC	M1=15	D2(X2,B2)				
BR	BCR	M1=15	R2				
NOP	BC	M1=0	D2(X2,B2)				
NOPR	BCR	M1=0	R2				
BH	BC	M1=2	D2(X2,B2)				
BHR	BCR	M1=2	R2				
BL	BC	M1=4	D2(X2,B2)				
BLR	BCR	M1=4	R2				
BE	BC	M1=8	D2(X2,B2)				
BER	BCR	M1=8	R2				
BNH	BC	M1=13	D2(X2,B2)				
BNHR	BCR	M1=13	R2				
BNL	BC	M1=11	D2(X2,B2)				
BNLR	BCR	M1=11	R2				
BNE	BC	M1=7	D2(X2,B2)				
BNER	BCR	M1=7	R2				
BP	BC	M1=2	D2(X2,B2)				
BPR	BCR	M1=2	R2				
BM	BC	M1=4	D2(X2,B2)				
BMR	BCR	M1=4	R2				
BZ	BC	M1=8	D2(X2,B2)				
BZR	BCR	M1=8	R2				
BO	BC	M1=1	D2(X2,B2)				
BOR	BCR	M1=1	R2				
BNP	BC	M1=13	D2(X2,B2)				
BNPR	BCR	M1=13	R2				
BNM	BC	M1=11	D2(X2,B2)				
BNMR	BCR	M1=11	R2				

Mnem.	Actual	Implied	Op1	Op2	Op3	Op4	Op5
BNZ	BC	M1=7	D2(X2,B2)				
BNZR	BCR	M1=7	R2				
BNO	BC	M1=7	D2(X2,B2)				
BNOR	BCR	M1=7	R2				
J	BRC	M1=15	RI2				
JLU	BRCL	M1=15	RI2				
JNOP	BRC	M1=0	RI2				
JLNOP	BRCL	M1=0	RI2				
JH	BRC	M1=2	RI2				
JLH	BRCL	M1=2	RI2				
JL	BRC	M1=4	RI2				
JLL	BRCL	M1=4	RI2				
JE	BRC	M1=8	RI2				
JLE	BRCL	M1=8	RI2				
JNH	BRC	M1=13	RI2				
JLNH	BRCL	M1=13	RI2				
JNL	BRC	M1=11	RI2				
JLNL	BRCL	M1=11	RI2				
JNE	BRC	M1=7	RI2				
JLNE	BRCL	M1=7	RI2				
JP	BRC	M1=2	RI2				
JLP	BRCL	M1=2	RI2				
JM	BRC	M1=4	RI2				
JLM	BRCL	M1=4	RI2				
JZ	BRC	M1=8	RI2				
JLZ	BRCL	M1=8	RI2				
JO	BRC	M1=1	RI2				
JLO	BRCL	M1=1	RI2				
JNP	BRC	M1=13	RI2				
JLNP	BRCL	M1=13	RI2				
JNM	BRC	M1=11	RI2				
JLNM	BRCL	M1=11	RI2				
JNZ	BRC	M1=7	RI2				
JLNZ	BRCL	M1=7	RI2				
JNO	BRC	M1=14	RI2				

Mnem.	Actual	Implied	Op1	Op2	Op3	Op4	Op5
JLNO	BRCL	M1=14	RI2				
JAS	BRAS	none	R1	RI2			
JASL	BRASL	none	R1	RI2			
JCT	BRCT	none	R1	RI2			
JCTG	BRCTG	none	R1	RI2			
JXH	BRXH	none	R1	R3	RI2		
JXHG	BRXHG	none	R1	R3	RI2		
JXLE	BRXLE	none	R1	R3	RI2		
JXLEG	BRXLG	none	R1	R2	RI2		

Appendix B – Embedding the Assembler

ASMA is designed for use as an embedded assembler. The `asma.py` Python script is simply a command-line interface that drives the assembler embedded within it. Embedding the assembler in a different module as a “back end” to some source of assembler statements will follow a design similar to that found in the `asma.py` module.

The assembler is actually the class `Assembler` found in the `assembler.py` module. A module embedding the assembler must import this module. Creating the `Assembler` object has three required positional arguments:

1. A string identifying the targeted CPU defined in the MSL database.
2. A string providing the path or file name of the MSL database file.
3. An `AsmOut` object identifying what output is to be created and where it is to be written.

Refer to the `Assembler.__init__()` method's arguments for details of optional parameters.

The user of the embedded assembler submits assembly statements to the assembler by means of the `statement()` method. The last submitted statement must contain the `END` directive. Statements are now queued for assembly.

Actually assembling of the statements is initiated using the `assemble()` method. When the assembly has completed, control is returned to the user of the embedded assembler. No output has been written. Results of the assembly are made available to the user via the `image()` method. This method returns an `Image` object that contains all of the requested output, all of the source statements processed by the assembler and accumulated `AssemblerError` exceptions. It is the responsibility of the using module to output the results of the assembly.

Refer to the `assemble.py` module for more details about theses mechanisms.