UNICOS System Information [2]

CAL, the Cray assembler, supports binary definition files and interfaces with the UNICOS operating system.

A typical interactive session involves assembling a CAL source file to create a relocatable object file. A link editor or loader processes the object file to create an executable file. This process is accomplished by entering a series of commands at the command line. (For more information, see subsection 2.2, page 19.)

CAL does not use the standard input file or standard output file during assembly; however, it does use the standard error file to report diagnostic and source line messages.

CAL generates listing and diagnostic messages during assembly. When the `-l` and `-L` options are specified on the `as`(1) command line and a syntax or semantic error is encountered, the assembler generates listing messages. A message is printed in the listing after each source statement flagged by the assembler and a pointer identifies the location within the source statement that corresponds to the message. The message also is issued to the standard error file.

CAL generates diagnostic messages that provide user information about the assembly (comment, note, and caution) and CAL assembler errors (warning and error). Diagnostic messages are classified by level of severity from low to high, as follows:

- User information about the assembly
  - Comment (statistical information)
  - Note (possible assembly problems)
  - Caution (definite user errors during assembly)
- CAL assembler errors
  - Warning (possible error such as truncation of a value)
  - Error (fatal assembly error, you should check the message and source code carefully for possible mistakes)

Note: To print comment-, note-, and caution-level diagnostic messages to the standard error file, you must specify the `-m` option on the `as`(1) command line.

as(1) – CAL command line

2.1

The UNICOS `as`(1) command invokes the CAL assembler. The format of the `as`(1) command is as follows:

```
as [-o objfile] [-l lstfile] [-L msgfile] [-b bdflist] [-B]
   [-c bdfile] [-D micdef] [-g symfile] [-G] [-C cpu] [-h] [-H]
   [-i nlist] [-I options] [-m mlevel] [-M] [-n number] [-f]
   [-F] [-j] [-J] [-U] [-V] file
```

The `as`(1) command assembles the specified file. The following options, each a separate argument, can appear in any order, but they must precede the *file* argument:

`-o` *objfile* — Relocatable assembly output is stored in file *objfile*. By default, the relocatable output file name is formed by removing the path name and the `.s` suffix, if they exist, from the input file and by appending the `.o` suffix. A link editor or loader must process *objfile*.

`-l` *lstfile* — Assembly output source listing is stored in file *lstfile*. By default, the output source listing is suppressed.

`-L` *msgfile* — Assembly output source message listing is stored in file *msgfile*. By default, the output message listing is suppressed.

`-b` *bdflist* — Reads the binary definition files stored in one or more files. The files specified in *bdflist* can be designated using one of the following forms:

- List of files separated by a comma
- List of files enclosed in double quotation marks and separated by a comma and/or one or more spaces

The files listed in *bdflist* are read in the order specified. By default, the binary assembler definitions found in file `/lib/asdef` are also read unless suppressed with the `-B` option.

`-B` Suppresses `/lib/asdef` as the default binary assembler definition file.

`-c` *bdfile* Creates the binary definition file *bdfile*. By default, the creation of a binary definition file is suppressed.

`-D` *micdef* Defines a globally defined constant micro *mname*, as follows:

micdef `::=` *mname*[=[*string*]]

mname must be a valid identifier. If the = character is specified, it must immediately follow *mname*. The *string* that immediately follows the = character, if any, is associated with *mname*. If you do not specify *string*, *mname* will be associated with an empty string.

If *mname* was defined as a micro by use of a binary definition file, the *mname* specified on the command line overrides the *mname* defined within the binary definition file; in that case, CAL issues a note-level diagnostic message.

`-g` *symfile* Assembly output symbol file is stored in *symfile*. *symfile* is used by the system debuggers. By default, the output symbol file is suppressed.

If you specify the same file for both the `-o` and `-g` options, and the last assembler segment does not contain a module (that is, it contains only the global part of the segment), CAL will not generate a corresponding symbol table for that assembler segment. For detailed information about segments, modules, and global parts, see section 3, page 33.

`-G` Forces all symbols to *symfile* if the `-g` option is used. Usually, nonreferenced symbols are not included.

`-C` *cpu* Generates code for the CPU specified. By default, code is generated for the machine specified by the `TARGET` environment variable. If the `TARGET` environment is not set, code is generated for the characteristics of the host machine. *cpu* has one of the following syntaxes:

cpu ::= *primary*{",["*charac*]}
or
cpu ::= ",["*charac*]{",["*charac*]}

primary *primary* can be one of the following Cray Research systems:

cray-c90	CRAY C90 series
cray-j90	CRAY J90 series
cray-ts	CRAY T90 series
cray-ymp	CRAY Y-MP series

charac Specifies the features of the *primary* computer.

Cray PVP systems permit you to specify the logical and numeric traits shown in Table 1.

Table 1. Logical and numeric traits

Traits	Description
Logical	
`avl`	Additional vector logical
`noavl`	No additional vector logical
`bdm`	Bidirectional memory
`nobdm`	No bidirectional memory
`bmm`	Bit matrix multiply (BMM), only on machines supporting BMM hardware.
`nobmm`	No bit matrix multiply
`cigs`	Compressed index and gather/scatter
`nocigs`	No compressed index and gather/scatter
`cori`	Control operand range interrupts
`nocori`	No control operand range interrupts
`ema`	Extended memory addressing
`noema`	No extended memory addressing
`hpm`	Hardware performance monitor
`nohpm`	No hardware performance monitor
`ieee`†	CRAY T90 system with IEEE floating-point hardware
`pc`	Programmable clock
`nopc`	No programmable clock
`readvl`	Read vector length
`noreadvl`	Do not read vector length
`statrg`	Status register
`nostatrg`	No status register
`vpop`	Vector pop count
`novpop`	No vector pop count

† The `ieee` characteristic can be used to specify that code be generated to run on a CRAY T90 system with IEEE floating-point hardware; however, generating code that runs on a Cray PVP system that uses Cray floating-point arithmetic from a CRAY T90 system with IEEE floating-point hardware is not supported.

Table 1. Logical and numeric traits
(continued)

Traits	Description
`vrecur`	Vector recursion
`novrecur`	No vector recursion
Numeric	
`bankbusy=`*n*††	Bank busy time (in clock periods)
`banks=`*n*††	Number of memory banks
`clocktim=`*n*††	Clock time (in picoseconds)
`ibufsize=`*n*††	Instruction buffer size (in words)
`memsize=`*n*††	Memory size (in words)
`memspeed=`*n*††	Memory speed (in clock periods)
`numclstr=`*n*††	Number of cluster registers
`numcpus=`*n*††	Number of CPUs

`-h` Enables all list pseudo instructions regardless of the location field name.

`-H` Disables all list pseudo instructions regardless of the location field name.

`-i` *nlist* Restricts list pseudo processing to those pseudo instructions whose location field names are given in *nlist*. The names specified by *nlist* can take one of the following forms:

- List of names separated by a comma
- List of names enclosed in double quotation marks and separated by a comma and/or one or more spaces

`-I` *options* Specifies a list of options. You can specify a list of more than one option without intervening blanks. You cannot specify conflicting options (for example, the same character in uppercase and lowercase) in the same `-I` list. For valid options, see Table 2.

†† *n* represents an unsigned decimal number

Table 2. List options

Options	Description
b	Enables macro, opdef, dup, and echo expansion (binary only)
B†	Disables macro, opdef, dup, and echo expansion (binary only)
c	Enables macro, opdef, dup, and echo expansion (conditionals)
C†	Disables macro, opdef, dup, and echo expansion (conditionals)
d	Enables dup and echo expansion
D†	Disables dup and echo expansion
e†	Enables edited statement listing
E	Disables edited statement listing
l	Enables listing control pseudo instructions
L†	Disables listing control pseudo instructions
m	Enables macro and opdef expansions (binary only)
M†	Disables macro and opdef expansions (binary only)
n†	Enables nonreferenced local symbols included in the cross-reference
N	Disables nonreferenced local symbols included in the cross-reference
p	Enables macro, opdef, dup, and echo expansion of pre-edited lines
P†	Disables macro, opdef, dup, and echo expansion of pre-edited lines

† Denotes default option

Table 2. List options
(continued)

Options	Description
`s`†	Enables source statement listing
`S`	Disables source statement listing
`t`	Enables text source statement listing
`T`†	Disables text source statement listing
`x`†	Enables cross-reference listing
`X`	Disables cross-reference listing

`-m` *mlevel*
: Specifies the level of the output listing, the message listing, and the standard error file. *mlevel* can be `comment`, `note`, `caution`, `warning`, or `error`.

  If you specify the `-m` option, it overrides all `MLEVEL` pseudo instructions. By default, the level is `warning`, and the `MLEVEL` pseudo instruction controls the message level during assembly.

`-M`
: Enables flagging of possible CRAY C90 series and CRAY J90 series bidirectional memory conflicts. Requires `-m` to be set to `comment`, `note`, or `caution`.

`-n` *number*
: Maximum number of messages that will be inserted into the output listing, the message listing, and the standard error file. *number* must be 0 or greater; the default is 100.

`-f`
: Enables the new statement format. By default, the old format is used when targeting for a CRAY Y-MP system; otherwise, the new format is used. Statement format reverts to the format that is specified on the invocation statement at the end of each assembler segment.

† Denotes default option

`-F`	Disables the new statement format. By default, the old format is used when targeting for a CRAY Y-MP system; otherwise, the new format is used. Statement format reverts to the format specified on the invocation statement at the end of each assembler segment.
`-j`	Enables editing; the default is enabled. Editing status reverts to the status specified on the invocation statement at the end of each assembler segment.
`-J`	Disables editing; the default is enabled. Editing status reverts to the status specified on the invocation statement at the end of each assembler segment.
`-U`	Forces the conversion of source code to uppercase. Quoted strings are embedded micros and are protected. Both new and old format statement types are supported.
`-V`	Causes the version number of the assembler being run and other statistical information (comment-level diagnostic messages) to be written to the standard error file.
file	File that will be assembled; all options must precede the file name argument.

Interactive assembly
2.2

To assemble and execute a CAL program interactively, enter the following commands:

```
as myfile.s

segldr myfile.o

a.out
```

The commands are described as follows:

Command	Description
`as`	Assembles file *myfile*`.s` and creates file *myfile*`.o`
`segldr`	Links and loads the assembled program found in *myfile*`.o` and creates the executable file `a.out`
`a.out`	Executes the executable file `a.out`

For a description of these and other commands, see the *UNICOS User Commands Reference Manual*, publication SR–2011.

The UNICOS environment
2.3

The following subsections describe aspects of the UNICOS environment. How the environment is set depends on the type of shell being used.

LPP *environment variable*
2.3.1

The CAL assembler is affected by the LPP environment variable in the UNICOS environment. The LPP environment variable sets the number of lines per page for output listings (page length). By default, the number of lines per page is 55.

To set the LPP environment variable and assemble multiple source files when using the C shell, enter the following commands:

```
setenv LPP n
as filenamea.s
as filenameb.s
.
.
.
```

To set the LPP environment variable and assemble a source file with a single command line when using the standard shell, enter the following command:

```
LPP=n as filenamex.s
```

If you specify the LPP shell variable on the same line as the as(1) command line, the number of lines per page assigned by the LPP shell variable is restricted to that particular as instruction.

To set the LPP environment variable and assemble multiple source files when using the standard shell, enter the following commands:

```
LPP=n

export LPP

as filenamea.s

as filenameb.s

.

.

.
```

If you specify the LPP shell variable as a separate entry and then export it, all assemblies that follow use the page length specified by that LPP shell variable for output and message listings.

In the preceding examples, *n* is a decimal number in a valid range of 4 through 999 (the default is 55) that represents the page length used in output listings and *filenamea, filenameb...* represent the names of the source files being assembled.

Note: If *n* is outside of the valid range, the page length is set to the default.

In the following example, the number of lines per page in the output listings for `srca.s` and `srcb.s` is 45:

```
LPP=45

export LPP

as srca.s

as srcb.s
```

In the following example, the page length for `srcd.s` is 45. However, the page length for `srce.s` reverts to 64 because the second `LPP` shell variable is restricted to the assembly of `srcd.s`:

```
LPP=64

export LPP

LPP=45 as srcd.s

as srce.s
```

`TMPDIR` *shell variable* 2.3.2

The `TMPDIR` shell variable specifies a directory used by the assembler for its temporary file. If the directory is not specified or is specified incorrectly, the assembler uses the system default. The default is site-specific.

To set the `TMPDIR` environment variable and assemble a source file when using the C shell, enter the following commands:

```
setenv TMPDIR dir_name

as filenamea.s
```

To set the `TMPDIR` environment variable and assemble a source file with a single command line when using the standard shell, enter the following command:

```
TMPDIR=p as filenamex.s
```

If you specify the `TMPDIR` shell variable on the same line as the `as`(1) command line, the temporary directory assigned by the `TMPDIR` shell variable affects only that particular `as` instruction.

To set the TMPDIR environment variable and assemble multiple source files when using the standard shell, enter the following commands:

```
TMPDIR=p

export TMPDIR

as filenamea.s

as filenameb.s

.

.

.
```

If you specify the TMPDIR environment variable as a separate entry and then export it, all assemblies that follow use the temporary directory specified by that TMPDIR shell variable for temporary files.

In the preceding example, *p* specifies the directory path used for the assembler's temporary file and the *filenamea, filenameb...* variables represent the names of the UNICOS files that are being assembled.

In the following example, /tmp is the directory that CAL uses for its temporary file:

```
TMPDIR=/tmp

export TMPDIR

as srca.s

as srcb.s
```

In the following example, the temporary directory used for `srcd.s` is the current working directory (`.`). The temporary directory for `srce.s`, however, reverts to `/usr/tmp`, which is used because the second `TMPDIR` environment variable is associated only with the assembly of `srcd.s`:

```
TMPDIR=/usr/tmp

export TMPDIR

TMPDIR=.  as srcd.s

as srce.s
```

`MSG_FORMAT` *error message format* 2.3.3

The `MSG_FORMAT` environment variable controls the format of error messages received from programs that use the `catmsgfmt`(3) message formatting routine. For more information, see the `explain`(1) man page.

`TARGET` shell variable 2.3.4

The `TARGET` environment variable determines the characteristics of the machine the code is generated for. To initialize the `TARGET` environment variable in the C shell, enter the following:

```
setenv TARGET cpuname
```

To initialize the `TARGET` environment variable in the standard shell, enter the following:

```
export TARGET
```

The format to set up or change the `TARGET` environment variable in the standard shell is as follows:

```
TARGET=[cpuname]{,[charac]}
```

If the `TARGET` environment variable is not set, code is generated using the characteristics of the host machine. The options for *cpuname* and *charac* may be found in subsection 2.1, page 12. For more information, see the `target`(1) man page.

Note: Targeting a CRAY T90 with IEEE floating-point hardware from any other Cray PVP system is supported, however; targeting a Cray PVP system that uses Cray floating-point arithmetic from a CRAY T90 with IEEE floating-point hardware is not supported.

Binary definition files

2.4

CAL allows the assembler source program access to previously assembled lines or sequences of code. These preassembled sequences are stored in files that are called *binary definition files*. Binary definition files are analogous to libraries and are one of the following types:

- System-defined
- User-defined

The system-defined binary definition file is `/lib/asdef`. CAL accesses the system-defined binary definition file automatically unless the assembler is directed otherwise. Binary definition files contain commonly used symbols, macros, opdefs, opsyns, and micros. For information about available macros and opdefs, see the *UNICOS Macros and Opdefs Reference Manual*, publication SR–2403.

Note: System- and user-defined binary definition files are identical in all respects. Both types of files are created and used in exactly the same manner. In this manual, they are treated as separate entities to encourage you to define binary definition files that meet your particular programming requirements.

You can create user-defined binary definition files by using either of the following methods:

- Copying the system-defined binary definition files and then modifying the new file either by adding new definitions or by redefining existing definitions.
- Disabling the recognition of system-defined binary definition files and accumulating the defined sequences entirely from an assembler source program. For more information, see subsection 2.1, page 12.

You can specify more than one binary definition file with each assembly. If more than one binary definition file is specified, the files are processed from left to right in the order specified by the `-b` option.

Lines or sequences of code assembled and stored in a binary definition file, can be accessed without reassembly. This means accessing a binary definition file directly saves assembler time.

The following subsections describe defining, creating, and using binary definition files.

Defining a binary definition file 2.4.1

Only certain types of lines or sequences of code are permitted in a binary definition file. Binary definition files are always created from the global part of program segments and from any currently accessed binary definition files. Typically, binary definition files are created from source programs that include one segment that contains a global part, but has no program module (see Figure 4, page 28).

Additions can be made to binary definition files from assembler source programs that include program modules, however, not all lines or sequences of code in the global part are added.

> **Note:** Under no circumstance is any line or sequence of code added to a binary definition file from an assembler program module. All additions to binary definition files come from the global part of the segment.

Binary definition files are composed of lines or sequences of code classified as follows:

- Symbols
- Macros
- Opdefs
- Opsyns
- Micros

Each line or sequence of code added to a binary definition file must be in one of these classes and must satisfy the requirements for that particular class.

Symbols
2.4.1.1

CAL accumulates the symbols to be included in a new binary definition file from all currently accessed binary definition files and from all of the global parts of program segments that fit the following requirements:

- Symbols cannot be redefinable.

  To be included in a binary definition file, a symbol must be defined with the = (equates) pseudo instruction. Symbols defined with the `SET` or `MICSIZE` pseudo instruction are redefinable; therefore, they are not included in a binary definition file.

- Symbols cannot be preceded by %%.

  This exclusion applies to symbols that are created by the `LOCAL` and = pseudo instructions.

CAL identifies all of the symbols in the global part of program segments that meet the preceding requirements and includes them in the creation of a binary definition file. In Figure 4, page 28, `SYM1`, `SYM3`, and `SYM4` meet the requirements and are included. `SYM2` (defined in the module), `SYM5` (redefinable), and `%%SYM6` (begins with %%) do not meet the requirements and are not included.

Macros
2.4.1.2

CAL accumulates the macros to be included in a new binary definition file from all currently accessed binary definition files and from all of the global parts of segments within a source program.

Opdefs
2.4.1.3

CAL accumulates opdefs (operation definitions) to be included in a new binary definition file from all currently accessed binary definition files and from all of the global parts of segments within a source program.

Opsyns
2.4.1.4

CAL accumulates opsyns (operation synonyms) to be included in a new binary definition file from all currently accessed binary definition files and from all of the global parts of segments within a source program.

Micros
2.4.1.5

CAL accumulates micros to be included in a new binary definition file from all currently accessed binary definition files and from all of the global parts of segments within source program. Only micros that cannot be redefined are included in a binary definition file. A micro must be defined using the `CMICRO` pseudo instruction to be included in a binary definition file.

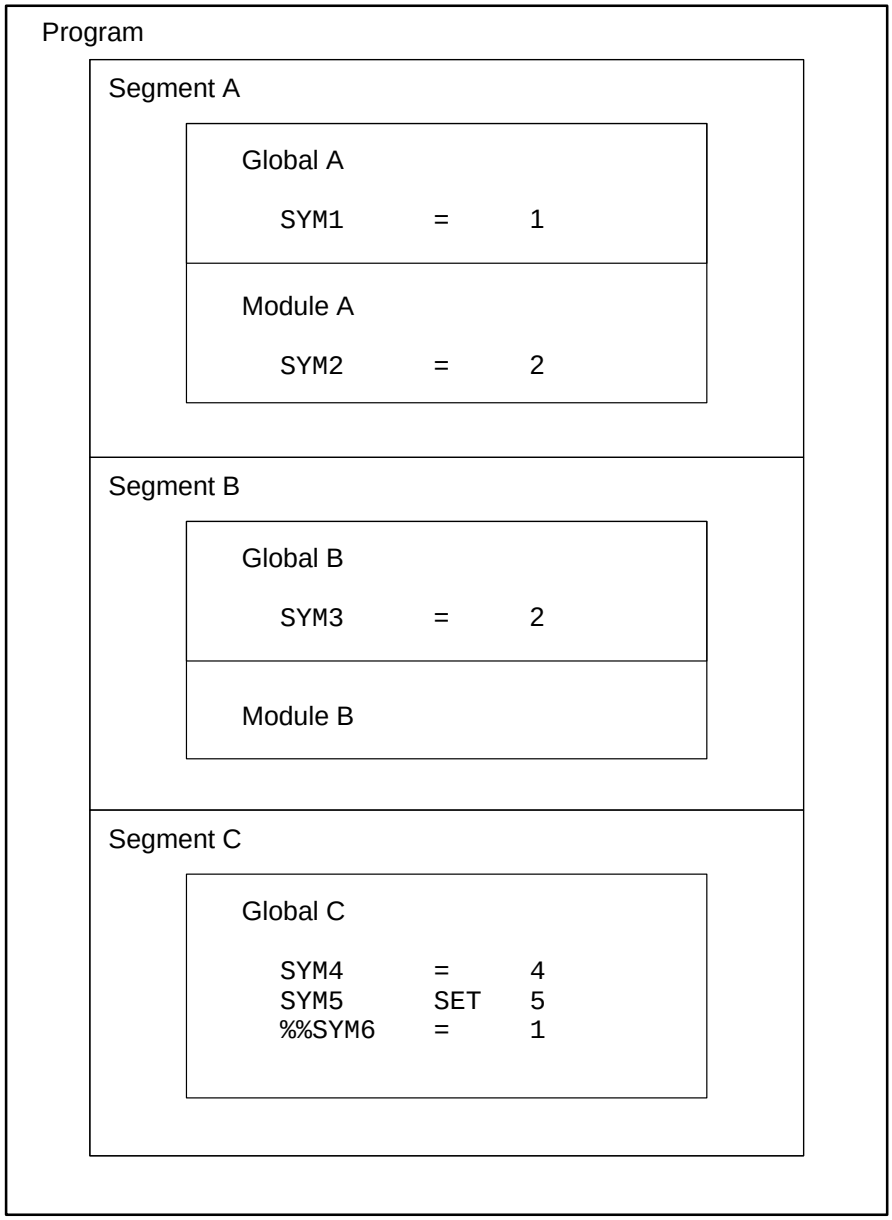

Figure 4. CAL program structure

Creating binary definition files 2.4.2

You can create your own binary definition files containing information related to both the UNICOS operating system and Cray Research hardware. Keep in mind that system dependencies included in these binary definition files may not be portable between UNICOS versions or different hardware platforms.

To create binary definition files under UNICOS, include the `-b` and `-c` options on the `as`(1) command line. The `-b` option accepts a list of files separated by commas or a list of files enclosed in double quotation marks and separated by spaces or commas as arguments.

In the following example, the default system-defined binary definition file `/lib/asdef` and user-defined binary definition files `ourdeffile` and `mydeffile` are included along with the accumulated symbols, macros, opdefs, opsyns, and micros from the global parts of the program segments from the current source program (`prog.s`) being assembled. The new binary definition file called `mynewfile` is defined and created by including the `-c` option.

```
as -b ourdeffile,mydeffile -c mynewfile prog.s
```

In CAL, the default binary definition file (`/lib/asdef`) is available unless suppressed by including the `-B` option. If not suppressed, `/lib/asdef` is the first binary definition file read. Any other binary definition files specified following the `-b` option are processed in the order specified. The following command line suppresses `/lib/asdef` and makes `mynewfile` the only available binary definition file:

```
as -B -b mynewfile prog.s
```

The following command line suppresses `/lib/asdef` and takes only the accumulated symbols, macros, opdefs, opsyns, and micros from the global parts of the program segments from the current source program being assembled and enters them into the binary definition file `mynewfile`:

```
as -B -c mynewfile prog.s
```

To use the system-defined binary definition file and specify the user-defined binary definition file created using the preceding options, use the following command line in subsequent assemblies:

```
as -b mynewfile prog.s
```

Using binary definitions files 2.4.3

Binary definition files provide access to previously assembled lines or sequences of code. To access binary definition files, include the `-b` option on the `as`(1) command line. When binary definition files are accessed, they are checked for the following:

- CPU compatibility
- Multiple references to the same definition

CPU compatibility checking 2.4.3.1

CAL permits access to any previously defined file with one restriction. Binary definition files are marked with the CPU type for which they were created. Binary definition files created on one Cray PVP system is not necessarily compatible with all Cray PVP systems. If a binary definition file is not compatible with the system you are using, the binary definition file is not accepted, and the following message is issued:

`Incompatible version of binary definition file '`*file*'

This check ensures that the machine on which the binary definition file was created is compatible with the program trying to use it. Some CAL pseudo instructions have restricted use based on hardware and software requirements. The binary definition file compatibility check prevents the mixing of binary definition files and ensures that hardware and software restrictions are not violated.

Multiple references to a definition 2.4.3.2

CAL checks for multiple references to definition names for macros and opsyns, location field names for symbols and micros, and syntax for opdefs. The following subsections describe how multiple references to a definition are resolved.

Symbols 2.4.3.2.1

If a symbol is defined in more than one binary definition file, the definitions are compared. If the definitions are identical, CAL disregards the duplicates and makes one entry for the symbol from the binary definition files. If a symbol is defined more than once and the definitions are not identical, CAL uses the last definition associated with the location field name and issues the following diagnostic message:

`Symbol '`*name*`' is redefined in file '`*file*`'`

Macros
2.4.3.2.2

If a macro with the same functional name is defined in more than one binary definition file, the definitions are compared. If the definitions associated with the macro's functional name are identical character by character, CAL disregards the duplicate definition and makes one entry for the macro from the binary definition files. If the functional name of the macro is used more than once, and the definitions associated with the functional name are not identical character by character, CAL uses the definition associated with the last reference to the functional name and issues the following diagnostic message:

`Macro` *'name'* `in file` *'file'* `replaces previous definition`

If a macro is defined with the same functional name as a pseudo instruction, the macro replaces the pseudo instruction and CAL issues the same message as shown above.

Opdefs
2.4.3.2.3

If an opdef with the same syntax is defined in more than one binary definition file, the definitions of the opdefs are compared. If the definitions of the two opdefs are exactly the same, CAL disregards the duplicate definition and makes one entry for the opdef from the binary definition files. If the same syntax appears more than once and the definitions are not exactly the same, the syntax associated with the last reference to the opdef is used as its definition and CAL issues the following diagnostic message:

`Opdef` *'name'* `in file` *'file'* `replaces previous definition`

If an opdef is defined with the same syntax as a machine instruction, the opdef replaces the machine instruction and CAL issues the message shown above.

Opsyn
2.4.3.2.4

If an opsyn with the same functional name is defined in more than one binary definition file, the definitions are compared. If the definitions are identical, CAL disregards the duplicate definition and makes one entry for the opsyn from the binary definition files. If the functional name for an opsyn is used more than once and the definitions are not identical, CAL uses the definition associated with the last reference to the opsyn name and issues the following diagnostic message:

`Opsyn` *'name'* `in file` *'file'* `replaces previous definition`

If an opsyn is defined with the same name as a pseudo instruction, the opsyn replaces the pseudo instruction and CAL issues the message as shown above. Pseudo instructions have an internal code that permits CAL to identify them when they are encountered. When an opsyn is used to redefine an existing pseudo instruction, CAL copies the predefined internal code of that pseudo instruction and uses it for identification in the binary definition file.

Micros
2.4.3.2.5

If a micro with the same location field name is defined in more than one binary definition file, the micro strings associated with the location field names are compared. If the strings are identical, CAL disregards the duplicate definition and makes one entry for the micro from the binary definition files. If the micro is used more than once and the strings associated with the micro names are not exactly identical, CAL uses the string associated with the last reference to the micro name and issues the following diagnostic message:

`Micro` *'name'* `in file` *'file'* `replaces previous definition`