Introl−CODE Reference

Introl Corporation

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Introl−CODE Reference

This document contains detailed descriptions of the command line programs supplied with CODE and file formats supported by CODE.

Two other documents are included with CODE. The first is the Chip Reference which describes the microcontrollers and microprocessors that are supported. The second is the Tcl/Tk Reference which describes the scripting language that comes with CODE.

Commands

Although CODE commands are normally run implicitly from the CODE development environment, you can also run CODE commands from the CODE command window, the system command prompt, or from a command script or **ibuild** Makefile.

You can run CODE commands immediately from the CODE command window. If you want to use CODE commands from the system command prompt, you have to set the **PATH** and **INTROL** environment variables properly. There is an explanation of this for Windows95/NT and the various Unix systems.

as05
: Assembler for the 68HC05.

as08
: Assembler for the 68HC08.

as09
: Assembler for the 6809.

as11
: Assembler for the 68HC11, 6801, and 6301.

as12
: Assembler for the 68HC12.

as16
: Assembler for the 68HC16.

as68
: Assembler for the 68332, and 68K family (68000, 68010, 68020, 68030, 68040, etc.).

cc09
: C compiler for the 6809.

cc11
: C compiler for the 68HC11, 6801, and 6301.

cc12
: C compiler for the 68HC12.

cc16
: C compiler for the 68HC16.

cc68
: C compiler for the 68332, and 68K family (68000, 68010, 68020, 68030, 68040, etc.).

icc
: The general C compiler. The ccXX forms above are equivalent to icc −gXX.

ild05
: Object code linker for the 68HC05.

ild08
: Object code linker for the 68HC08.

ild09
: Object code linker for the 6809.

ild11
: Object code linker for the 68HC11, 6801, 6301.

ild12
: Object code linker for the 68HC12.

ild16
: Object code linker for the 68HC16.

ild68
: Object code linker for the 68332, and 68K family (68000, 68010, 68020, 68030, 68040, etc.).

ild
: The general linker. The ildXX forms above are equivalent to ild −gXX.

codesh
: The Tcl command interpreter with integrated CODE extensions.

codewish
: The Tcl command interpreter and Tk graphical toolkit with integrated CODE extensions.

idb
: The command line/character window debugger for all targets.

iadr
: Create a listing file containing the program source and the addresses where the compiled code resides after linking.

iar
: Create and modify a library of files.

ibuild
: Build a program (Like *make*).

idbg
: Create Pentica DBG files from an ICOFF file.

idump
: Display the contents of an ICOFF file in human readable form.

ihex
: Convert ICOFF files into several hexadecimal file formats, such as S−records and Introl hex.

ihp
: Convert ICOFF files into HP64000 emulator format.

imerge
: Create a merged C and assembly language source listing.

ipe
: Convert ICOFF files into P&E Microsystems debug files.

isym
: Create a user defined text file contaning symbol information in an ICOFF file.

i695
: Convert ICOFF files into IEEE695 debug files.

The Assemblers

This chapter instructs on using the assemblers supplied with Introl−CODE. The included assemblers (and the processors they support, the default is in **bold**) are:

Assembler	Supported Processor(s) (default is **bold**)
as05	**68HC05**
as08	**68HC08**
as09	**6809**
as11	**68HC11**, 6801, 6301
as12	**68HC12**
as16	**68HC16**
as68	**683XX**, 68000, 68010, 68020, 68030, 68EC030, 68040, 68EC040

Assembler command line
: How to run the assemblers.

Processor type selection
: Selecting the specific processor for those assemblers that support a family of processors.

Substitutions
: How macro and command line parameter substitutions are done in the source file.

Macros
: How to create and use the assemblers' macro facilities.

Conditional assembly
: How to make the assembler assemble code based on input conditions.

Expressions
: How to make and use assembly and link time expressions.

Format of input files
: What an assembly language source file should look like.

Format of output files
: How files created by the assemblers look.

Directives
: Definitions of the assembler directives supported by the assemblers.

Error messages and recovery
: Error messages produced by the assemblers: what they mean and how to correct the errors.

Assembler command line

Assembler commands lines consist of the assembler name, zero or more assembler options, and the name of a source file to assemble.

The section Format of input files describes the assembly language syntax accepted by the assemblers.

```
as05 [{options}] filename

as08 [{options}] filename

as09 [{options}] filename

as11 [{options}] filename

as12 [{options}] filename

as16 [{options}] filename

as68 [{options}] filename
```

By convention, the name of assembly language files have the extension **.s** or **.s*XX***, where the *XX* is replaced with the two digits in the assembler's name, for example, **.s05** for the 68HC05.

Output files produced by the assembler are given names based on the original file name unless overridden by command line options below. The assembler will replace the source file's extension with an extension appropriate to the output file type. For example, if the original source file is **test.s09**, the listing file will be placed in the file **test.lst**, and the object file will be placed in the file **test.o09**. The section Format of output files describes the object and listing files produced by the assemblers.

Assembler options are precedded with a dash (`−'). The following options are available in Introl assemblers:

−a
: Directs the assembler to put static equates and symbols whose names begin with a question mark into the symbol table. These symbols are normally left out to save disk space in the resultant object file.

−b
: Include conditional assembly directives in the listing file. By default, the listing file contains lines that are assembled, not lines that contain directives controlling conditional assembly.

−c
: Direct the listing file to standard output instead of ***filename*.lst**.

−d
: Do not put macro definitions in the listing file. By default, macro definitions are included in the listing file.

−e
: Do not put macro calls in the listing file. By default, macro calls are included in the listing file.

−E
: Print errors not directed to the listing file in a standard format.

−f
: Direct the assembler to include all source code in the listing file, whether it was assembled or not (see conditional assembly). By default, the assembler only lists code that has been assembled in the listing

file.

−g

Direct the assembler to use continuation lines in the listing file for any directive or instruction that generates more than eight bytes of object code.

−gproc

Define the processor for which the assembly program is written. For those assemblers that support multiple similar processors this option is used to select a specific processor. This option performs a function similar to the opt directives that select a processor.

Option	Assembler	Processor
−g05	as05 default	68HC05
−g08	as08 default	68HC08
−g09	as09 default	6809
−g11	as11 default	68HC11
−g01	as11	6801, 6803
−g03	as11	6301, 6303 (Hitachi)
−g12	as12 default	68HC12
−g16	as16 default	68HC16
−g68	as68 default	68332 family
−g00	as68	68000
−g10	as68	68010
−g20	as68	68020/68030
−g30	as68	68030
−gec30	as68	68EC030
−g40	as68	68040
−gec40	as68	680EC40
−g881	as68	68881 FPU

−h

Prevent the assembler from attempting to open a file named ***opcode*.mac** when encountering an undefined macro named ***opcode***. The default behavior is to treat an undefined opcode as a file macro.

−i

Put the listing of all files included with the **lib** directive into the listing file. Normally, included files are not part of the listing file.

−j

as68 only. Inhibits the optimization of **move.l** and **add.l** into **moveq** and **addq**. Normally this optimization is performed. See also, the **ov** option on the **opt** directive.

−k

Print the name and version of the assembler during execution.

−l

Add information to the object file for each assembly language line that produces object code. This option allows you to do source−level debugging of assembly language programs.

−l=filename

Set listing file name to ***filename***.

−m

Print all the code generated by a macro expansion in the listing file. The default is to print the macro call only.

−n

Inhibit the insertion of the assembly output in the listing file. The symbol table and cross−reference will still be printed. Since **−s** inhibits the symbol table and **−r** inhibits the cross−reference, **−nsr** will inhibit the entire listing file.

−o=filename

Set the name of the object file to ***filename*.o68**.

−pinteger=string

Set a command line macro. ***integer*** can be between 0 and 99. ***string*** replaces any positional parameters that occur in the source text. (see Substitutions) In the simplest case,

```
as12 -p0=foo test.s12
```

will cause any occurances of **\0** (backslash zero) in the file **test.s12** to be replaced with **foo**.

−P

Read the assembly source from standard input. *filename* is required if there is no **−o** option to allow the assembler to generate an appropriately named object file.

−q

Disable pagination in the listing file.

−r

Inhibit the insertion of the cross−reference in the listing file. To completely inhibit the listing file, use **−nsr**.

−s

Inhibit the insertion of the symbol table symbol table in the listing file. To completely inhibit the listing file, use **−nsr**.

−t

Prevent the assembler from performing branch size optimizations. Normally the assembler will replace conditional and unconditional branch and jump instructions with equivalent instructions that are the smallest possible and yet are still able to reach the target of the branch or jump.

−u

Implicitly import all undefined symbols. Undefined symbols will normally cause the assembler to report an error. With this option, all undefined symbols are implicitly treated as if an **import** directive had been seen listing all undefined symbols. If these symbols remain undefined at link time then the linker will issue an error message.

−v

Use line feeds instead of form feeds in the listing file.

−x

Do syntax and semantic checking only, don't create an object file.

−y=path

Add ***path*** to the beginning of the list of paths searched for file macros and files specified by the **lib** directive. The current directory is searched first before the list.

−yy=path

Add ***path*** to the beginning of the list of paths searched for file macros and files specified by the **lib** directive. The current directory is searched last after the list.

−z

Delete the assembler's input file when the assembler is finished with it. This option will delete your original source file! This option is used by compilers to delete the compiler created assembly language source file.

Processor type selection

CODE contains assemblers for many Motorola microcontrollers. The assemblers, and the processors they support are listed in the folowing table.

Assembler	Supported Processor(s) (default is **bold**)
as05	**68HC05**
as08	**68HC08**
as09	**6809**
as11	**68HC11**, 6801, 6301
as12	**68HC12**
as16	**68HC16**
as68	**683XX**, 68000, 68010, 68020, 68030, 68EC030, 68040, 68EC040

The Introl−CODE assemblers **as11** and **as68** can assemble code for several processors, each with a slightly different instruction set.

The target processor and math coprocessor (if any) supported by the assembler are selected by using the **opt** directive.

The **opt** directive specifying the specific target processor must preceed any instructions to be assembled.

For example, in **as68**, to specify the 68020 microprocessor with the 68881 math co−processor, the following **opt** directive would be used:

```
        opt     cpu_68020,fpu_68881
```

Substitutions

Substitutions can pass parameters into a file from the assembler command line or pass parameters into a macro from a macro call. The backslash (`\') character introduces an *escape sequence* which is replaced by text as indicated in the following table. The listing file will show the replacement text−not the original text. An escape sequence not listed below is ignored and will cause the backslash to be removed from the sequence. In such a case, the escape sequence becomes a regular character.

Escape sequence	Replacement text
digit	parameter number *digit*
\[*integer*]	parameter number *integer*
\c	parameter count
\m	opcode modifier
\o	opcode name

\r	repeat count
\?−n or \?+n	internal label
\.	macro label
\{*symbol*}	*symbol* value (in textual form)

Each of the escape sequences are described in the following paragraphs. There are two possible contexts in which any of the sequences may be expanded: outside of a macro expansion or during a macro expansion, depending on its context.

Outside of a macro expansion, *digit* or \[*integer*] refers to the argument of a **−p** command line option; the first parameter, **\0**, is given by the **−p0** option, the second, **\1**, by **−p1**, and so on. During a macro expansion, these escape sequences refer to the operands of the macro being expanded.

Each of the examples given below assumes a file named *test.s* that is assembled with the command line:

`as68 test.s −p0=5 −p1=x` ***digit***, where ***digit*** is 0..9, is replaced with parameter number ***digit*** with the first parameter being number 0. This allows access to the first 10 parameters. The **\[*integer*]** form is used to access up to 100 parameters. For example, the following file:

```
                section         .data
                dc.b            \0+10
```

when assembled as described above would expand to:

```
                section         .data
                dc.b            5+10
```

Inside a macro, **\c** is replaced with the number of parameters present. Outside of a macro, it is replaced with a value that is one greater than the largest **−p** option given. For example, the following file would generate the value 2 (since the largest **−p** option was **−p1**):

```
                section         .data
                dc.b            \c
```

\m is replaced with the single−character type modifier of the invoking instruction or macro. If there is no type modifier character, a space is used. For example, the following:

```
test            macro
                dc.\m           \0
                endm
                test.b          5
                test            10
```

would expand to:

```
                dc.b            5
                dc              10
```

\o is replaced with the name of the source file being assembled when occurring outside of a macro expansion. During a macro expansion, **\o** is replaced with the name of the macro being expanded. For example, the following:

```
abc             macro
                dc.b            '\o'
                endm
                dc.b            '\o'
                abc
```

would expand to (remember that the file name is *test.s*):

```
                dc.b            'test.s'
                dc.b            'abc'
```

\r is replaced with the current repeat count. If a repeat directive is not currently in force, **\r** is replaced with a zero. For example, the following assembler file will generate the byte values from 0 to 10:

```
                section         .text
                repeat          0,10
                dc.b            \r
```

\?+*n* and **\?−*n*** expand to a question mark followed by the value of a counter which is incremented every time a **\?** appears in the label field of an assembler source line. The optional +*n* and −*n* forms expand to a question mark followed by, respectively, the *n*th subsequent or the *n*th previous value of the same counter.

The **\?+*n*** and **\?−*n*** forms are generally used in operand fields to refer to labels that have been previously generated using the **\?** form. For example, the following program:

```
                section         .data
label\?         dc.l            5
label\?         dc.l            \?              ; Use this label
label\?         dc.l            \?−2            ; Use the 2nd prev. label
label\?         dc.l            \?+1            ; Use the next label
label\?         dc.l            50
```

would expand to:

```
                section         .data
label?1         dc.l            5
label?2         dc.l            label?2         ; Use this label
label?3         dc.l            label?1         ; Use the 2nd prev. label
label?4         dc.l            label?5         ; Use the next label
label?5         dc.l            50
```

\. is replaced with `1' when occurring outside the context of a macro expansion. During a macro expansion, it expands to a question mark followed by the value of a counter which is incremented each time a macro is expanded. The initial value of the counter is one and it is incremented immediately after a macro is recognized. This means that the lowest accessible value of the counter is two. For example, the following fragment:

```
test            macro
label\?         dc.b            1
                endm

                section         .data
                test
                test
```

would expand to:

```
                section         .data
label?2         dc.b            1
label?3         dc.b            1
```

\{*symbol*} is replaced with the value of the symbol named *symbol*. The symbol must be absolute since its value is needed at assembly−time. For example, the following:

```
sym1            equ             5
sym2            equ             10
sym3            equ             sym1*sym2

                section         .text
label\{sym3}:
                ...
```

would create a global label named *label50* at the beginning of the *.text* section.

Macros

Macros are names which you define to extend the instruction set (both opcodes and directives) of the assembler. They are used in the place of a machine instruction or directive which the assembler replaces with the macro body text. Macros must always be defined before they are used otherwise phasing errors will result. For example, the following would define a macro called *make5* which could be used just like an opcode and has the effect of generating the constant 5:

```
make5           macro
                dc.b            5
                endm
```

Macros in the assembler may be of two types:

- *Normal macros*, which are defined using the **macro** and **endm** directives.
- *File macros*, which are defined by creating a text file with the name of the macro and a file name extension of *.mac*.

File macros are recognized by the assembler when a name occurs in the opcode field of a source file line that is neither a predefined opcode nor a previously defined macro name. The assembler will take the name in the opcode field, add a *.mac* suffix to it, and attempt to open the resultant name as a file. The contents of the file will then replace the source line in the assembly process. File macros, therefore, must be contained in files having a *.mac* file name extension.

File macros have the same effect as the **lib** directive with one exception: file macros can accept parameters. A common use for file macros is to define a non−file macro that has the same name as its original file macro name. This allows macros to be defined on an as−needed basis without needing to explicitly maintain a list of defined macros in the source file. For example, such a self−defining file macro for the previously described *make5* macro could be created using the following file macro (note: assume in this example of a file macro that the text is contained in a file named *make5.mac*):

```
make5           macro                   ; define a macro named `make5'
                dc.b            5
                endm

                make5                   ; invoke the macro
```

In this case, the macro *make5* will first cause the assembler to read the contents of the macro file which will

both define a non−file macro named *make5* and then invoke the newly defined non−file macro. If a macro taking arguments were being defined this way, the initial invocation at the bottom of the file would have to use the backslash (`\') notation for passing them (See Substitutions).

Parameters

Macro *parameters* are strings separated by commas and placed in the operand field of the macro invocation line. The body of the macro may then contain parameter substitution operations which will substitute the text of the parameters into the text of the expanded macro. This allows the same macro to be expanded many different ways depending on the macro definition and the parameters passed to the macro expansion.

Macro parameters that are enclosed in single or double quotes are passed as a single string parameter and may contain spaces and and commas.

Defining non−file macros

The definition of a normal (non−file) macro consists of body text placed between the macro and the endm directives as follows:

```
name            macro                   [{arg{,arg}}]
                ...                     ; body of macro
                endm
```

where:

- *name* is the name of the macro; and
- [{*arg*{,*arg*}}] are optional dummy parameters that may be used to allow *symbolic* parameter substitution.

For example, two ways of defining the same macro are:

```
add3            macro
                move.\m         \0,d0           ; read the first parameter
                add.\m          \1,d0           ; add the second
                move.\m         d0,\3           ; and save in the third
                endm
```

and:

```
add3            macro           first,second,result
                move.\m         first,d0        ; read the first parameter
                add.\m          second,d0       ; add the second
                move.\m         d0,result       ; and save in the third
                endm
```

In both of the examples above, if the macro is called with the line:

```
                add3.w            label,#10,to
```

the expanded code will be:

```
                move.w          label,d0        ; read the first parameter
                add.w           #10,d0          ; add the second
                move.w          to,d0           ; and save in the third
```

Numeric positional parameters can be mixed with symbolic substitution in the same definition. For example,

in the above example with symbolic parameters, the third parameter could have been accessed by either the symbol ***result*** or by the positional parameter **\2**. The symbolic parameter form has the advantage of readability; the positional parameter form has the advantage of flexibility. Positional parameter substitution may be done anywhere in the source file, including the middle of labels and strings. Symbolic substitution, however, may only be done where the macro processor can recognize the symbol. For example, a part of a macro definition may be:

```
                        ...                     ; other macro text
lab\0                   move.w          temp,d0 ; make a label by
*                                               ; concatenating "lab" and the
*                                               ; first positional parameter
                       ...
```

However, string concatenation is impossible with symbolic parameters:

```
                         ...                    ; other macro text

labresult                move.w         temp,d0
*                                               ; the label is always "labresult"
                         ...                    ; even if a "result" parameter
*                                               ; exists
```

Substitutions within macros

As mentioned earlier, the assembler performs various substitutions. Although these substitutions seem to have an immediate effect, (i.e. if the assembler sees **\0**, it immediately substitutes the text defined in parameter 0) this is not always true. Substitutions are suppressed between the macro and the endm directives and also during the reading of a file macro. For example, if a macro contains an instance of **\0** the two characters \0 are actually part of the macro's definition, rather than being immediately substituted.

It is also possible to access the parameters of an outer macro in the case of a macro being expanded during the definition of a macro. This is indicated by placing any number of ^ characters following the \. Each ^ indicates that one level of context should be skipped. It is possible to access parameters specified with the −p command line option this way (using enough ^ characters). For example the following:

```
inner           macro
                dc.b            \^0
                endm

outer           macro
                inner           5
                endm

                section         .data
                outer           10
```

would generate the constant 10 (not 5 − the `5' is not used). The **\^0** in the definition of inner tells the assembler to use the first parameter from the calling context of the macro. It's important to realize that this type of parameter expands differently depending on the context in which the macro was expanded. For example, if the **.data** section of the previous example contained:

```
                section         .data
                outer           10
                inner           100
```

the inner macro would generate the constant indicated by the **−p0** command line option (the `100' is not used).

The exitm directive

The **exitm** directive terminates macro source statement generation during expansion. It may be used within a conditional assembly structure to cause the assembler to ignore any subsequent source lines up to the **endm** directive. The **exitm** directive will also terminate any remaining conditional assembly structures that may be pending within the macro currently being expanded. For example:

```
fixdat          macro           arg
                ifc             arg,""
                err             argument expected in fixdat
                exitm                   ; abort macro expansion if no arg
                endif
                ...                     ; the rest of the macro body
                endm
```

Conditional assembly

The assembler supports *conditional assembly* of source statements, allowing sections of code within a source file to be included or excluded when the file is assembled. Assembly−time conditions may be specified through the use of arguments in the case of macros, and also via symbol definitions using the equ and set directives. Conditional assembly can also be controlled with the **−p** command line option.

The general structure of a code section which is to be conditionally assembled is of the form:

```
                if      abolute_expression      ; or ifn, ifc, etc.
                ...                             ; the code here is assembled
                ...                             ; if the condition is true
                else                            ; optional "else"
                ...                             ; the code here is assembled
                ...                             ; if the condition is false
                endif
```

where the **if** may be any of the conditional directives.

When an **if**, directive is encountered, the specified condition is evaluated to determine whether the resultant conditional statement is true or false. If the conditional statement is true, the code between the directive and the following **else** or **endif** directive will be assembled. If the optional **else** directive is used, the code between the **else** and the following **endif** is assembled if the conditional statement is false.

The *expression* in a conditional assembly directive must be absolute, that is, the assembler must be able to calculate the value of the expression at assembly time, see Expressions.

Code subject to conditional assembly may contain conditional assembly structures. Such nesting is permissible to a depth of 10 levels. It is important to note, however, that each nested conditional structure must be terminated with its own unique **endif** directive. A conditional assembly structure within a macro will also be terminated by the **exitm** directive, as mentioned earlier.

When exiting from a macro, all conditional assembly structures which have originated within that macro will always be automatically terminated. Conditional assembly structures may be terminated within a macro only if they have also originated within that same macro.

Expressions

Expressions containing symbols defined in a relocatable section are called *relocatable expressions*. Expressions containing no such symbols are called *absolute expressions*. A relocatable expression can be considered the more restrictive of the two since there are some contexts in which an expression's value must be known by the assembler. For example, the operand to the **repeat** directive must be an absolute expression. Any context in which a value must be known at assembly time will require an absolute expression. Symbols and constant values can be used interchangeably in an expression. All results of an expression at assembly time are 32−bit, truncated integers.

Operators

+	unary plus
−	unary negate (two's complement)
~	not (one's complement)
*	multiplication
/	division
%	modulus (remainder)
+	addition
−	subtraction
<<	shift left
>>	shift right
&	bitwise and
^	bitwise exclusive or
|	bitwise inclusive or
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
==	equal to
!=	not equal to

Precedence of the operators is, from highest to lowest:

1. − + (unary) ~
2. * / %
3. + −
4. >> <<
5. < > <= >=
6. == !=
7. &

8. ^
9. |

Parentheses are allowed in expressions to change the precedence of an operator. Assembly time expressions can be used in the operand of any assembler opcode or directive.

Constants

Constant values are defined as a numeric digit (0..9), followed by zero or more numeric digits or the letters A..F or a..f, followed by a radix indicator:

N*radix* where N is 0..9, A..F, or a..f (must be a valid digit in the given radix), preceded by a numeric digit, and *radix* is:

H
: hexadecimal

O,Q
: octal

B
: binary

D or nothing
: decimal

An alternate way of specifying constants is by preceding the constant by the alternate radix indicator followed by one or more valid digits in the given radix:
*alt_radix*N where *alt_radix* is:

$
0x
0X
: hexadecimal

@
: octal

%
: binary

& or nothing
: decimal

and N is 0..9, A..F, or a..f (must be a valid digit in the given radix). No preceding numeric digit is required.

Constants may also be ASCII character constants, in single quotes.

The assembler also recognizes a special constant that represents the current offset in the section: **$** or *****. When **$** or ***** is used in an expression, the value taken is the current offset in the section at the beginning of the line containing the **$** or *****. This constant, also known as the *current location counter*, has an absolute value if the current section is absolute and a relocatable value if the current section is relocatable.

Symbols

Non−local symbols

Non−local symbols are made up of letters (a..z, A..Z), digits (0..9), question marks (`?'), dollar signs (`$'), underscores (`_'), and periods (`.'). They must begin with either a letter, a period, or a question mark, and can be any length (only the first 16 characters of the symbol name will be printed in the listing file). A non−local

symbol is one that may be referenced by name and may be exported, or made visible in other source files, with the **export** directive. If it is exported, it may be imported into another source file with the **import** directive and referenced.

Symbol names are case sensitive, so *abc* is not the same as *Abc*.

Local symbols

A *local symbol* is a symbol which may be redefined many times, and is referenced by concatenating an **s** or an **l** to its name. They may not be exported. The name of a local symbol consists entirely of digits. The previous definition of a local symbol may be referenced by concatenating an **s** (for ``sooner'') to the symbol's name, while the next definition of a local symbol may be referenced by concatenating an **l** (for ``later'') to the symbol's name.

Colon functions

These functions can be used in expressions like integer constants or symbols; they are semantically equivalent to integer constants.

:defined(symbol)
: Returns 1 if *symbol* is defined, 0 if not.

:imported(symbol)
: Returns 1 if *symbol* arg is imported, 0 if not.

:absolute(symbol)
: Returns 1 if *symbol* is absolute, 0 if not.

:hasinstruction(inst)
: **as11 and as68 only.** Returns 1 if the target has the instruction *inst*, 0 if not.

:cpu_processor()
: **as11 and as68 only.** Returns 1 if *processor* is the current processor, 0 if not. The processor is set using the **opt** directive.

:fpu_processor()
: **as68 only.** Returns 1 if *processor* is the current floating point processor, 0 if not. The floating point processor is set using the **opt** directive.

Examples

The following constants all represent the same value:

10
: Decimal;

$a
: Hexadecimal;

%1010
: Binary;

12Q
: Octal.

The following are legal symbols:

abc
: Begins with a letter and contains only letters.

.a1

Begins with a period and contains only letters and digits (usually symbols that begin with a period are used for section names).

_a5

Begins with an underscore and contains only letters and digits.

The following are illegal symbols:

1abc

Cannot begin with a digit;

a]b

Cannot contain a square bracket.

The following examples of expressions assume the assembler source file:

```
foo             equ             5
                section         .code
label           nop
label1          nop
```

An *absolute expression* does not contain any symbols defined in a relocatable section. The following expressions are absolute:

10

Integer constants are absolute.

foo

A symbol defined with equ.

foo+10

An expression containing only integers and absolute symbols.

label1−label

The difference between two relocatable symbols defined in the same section is absolute.

The following expressions are *relocatable*:

label

Relocatable symbols are relocatable expressions.

label+10

Expressions containing at least one relocatable symbol that is not cancelled out by subtraction (see above) are relocatable.

Format of input files

The assembler expects its input to be an ASCII file which contains assembler text. Each source line is either a line of the following format:

[*symbol*][**:**] [*opcode* [*operand*{**,***operand*}]] [**;***comment*] or a line beginning with an asterisk (`*') or semicolon (';'), indicating a comment line. Comments may be placed at the end of a source line; they may be separated from the opcode or operand field by either a space or a semicolon. A source line containing only a symbol but no opcode or operand field can only be commented by separating the comment with a semicolon; otherwise the assembler will interpret the first word of the comment as an opcode.

Any *whitespace* (spaces or tabs) can separate the opcode field from the symbol field, the operand field from the opcode field or the comment field from the operand field. Since the assembler uses any space as its separator character, expressions used in the operand field may not contain spaces.

An optional colon after the symbol name in the first field causes a symbol to be made *global*, or implicitly exported (see the export directive).

We recommend that you use a semicolon to separate comments from the rest of source lines (even though it is optional). A future version of the assembler will have a command line option that will allow whitespace to be used more freely in source lines, but will demand a semicolon as a comment separator. Using semicolons now will make your source files compatible with this mode when it becomes available.

The opcode field of a source statement consists of either an assembly directive, an instruction, or a macro call. Opcodes may be either upper or lower case.

The maximum length of an assembly language source line is 10240 characters. There is no line continuation character.

The maximum length of a symbol is limited by the maximum line length and the first 100 characters are significant.

Format of output files

Relocatable object files

The assembler produces *relocatable* object code placed into modules called sections. These sections have no fixed memory addresses associated with them when they are assembled, but have addresses determined at link time.

To clarify the operation of the assembler, consider the following complete assembler source file:

```
                section                 .text
                jmp                     label
                nop
label           nop
```

This source file tells the assembler to create the object code for the **jmp** instruction as well as a pair of **nop** instructions in a section named *.text*. The object code produced by this program is (in hexadecimal):

```
                4E F9 XX XX XX XX 4E 71 4E 71
```

The bytes indicated by *XX* represent the address of label named *label*. The value of the *XX* bytes is indeterminate at assembly−time because the assembler does not know where the beginning of the section named *.text* will actually be located (note that the assembler actually generates bytes of zeroes for these indeterminate addresses). This explains some of the extra information stored in a relocatable object file (by extra information, we are referring to data stored in an object file other than that produced by your assembler code); namely, the *relocation records*. These records are added to the object file to tell the linker where to plug in final values. In this case, there would be a relocation record produced that told the linker to insert the value of *label* at offset in the file as a word. The other major set of additional information placed in a relocatable object file is the s*ymbol table* which correlates symbol (i.e. labels) with their values. For a complete description of Introl's object file format, see ICOFF.

Absolute object files

Introl's linker, by default, produces an *absolute* object file which is very similar to the previously described relocatable object files except that all of the sections have been bound to absolute addresses and all undefined symbol references have been resolved. The code and data in the file can be converted to hexadecimal and directly loaded into your target system and executed. The linker can, optionally, produce a relocatable object

file to be used, for example, as input to another run of the linker.

Even though the Introl assembler is a relocating assembler, it can also generate an absolute object file. If the assembler source file does not reference any undefined symbols and does not contain any relocatable sections (i.e. all code and data is defined using the org directive or the section directive with the addr qualifier), it will produce an absolute object file for which no linking is necessary.

Listing files

The assembler produces a *listing file* containing the following parts in order:

- source listing;
- section synopsis;
- symbol table; and
- cross−reference.

Any errors that occurred during assembly are displayed in the source listing following the offending line. The name of the listing file is derived from that of the source file by substituting the **.s*XX*** extension with **.lst**.

Source lines

The source listing contains the source lines combined with the machine code generated by that line. Lines in the first part have the following format:

```
AAAAAAAA BBBBBBBB ZCCCCCCCCCCCCCCCC XXXX?
```

- Field A is the source line number;
- Field B is the address of the first byte of code produced;
- Field C is the machine code produced;
- Field X is a copy of the source line;
- Field Z contains a right arrow (>) if a relocation record was generated.

Comment lines

Comment lines appear in the listing file exactly as they appear in the source file. They are aligned with the first column of other source lines, beginning at Field X.

Error messages

Error messages have the following form:

```
**** filename − line_number error_message
```

Symbol table

The second part of the listing file contains the *symbol table*. Each symbol referenced in the assembly source file is given a symbol table entry.

Symbol table entries are printed in as many columns that can fit in the current line length. They are sorted alphabetically by name from the top of the page to the bottom. Each symbol table entry has the following form:

```
AAAAAAAAAAAAAAAAB C DDD EEEEEEEE
```

- Field A is up to 16 characters of the symbol's name;
- Field B is a plus sign (+) if the symbol?s name is longer than 16 characters;
- Field C is a flag character that is **E** if the symbol is exported, **I** if it is imported, **U** if it is undefined, or a space if it is not global;
- Field D is the section number corresponding to the section in which the symbol is defined;
- Field E is the value of the symbol in hexadecimal. For comm symbols, the value represents the size of the common block.

For example, a two column symbol table might look like:

```
.bootvec                2 00000000 | __exit             E       1 00000058
.text                   1 00000000 | __initend          I       0 00000000
.vectors                3 00000000 | __initstart        I       0 00000000
```

Only the first sixteen characters of a symbol are printed in the symbol table listing.

Cross−reference

The third part of the listing contains a *cross−reference* of the symbols that have been defined or used. The cross−reference table is sorted by symbol name and one symbol per line is printed, with continuation lines if there are many references. The format of the cross−reference is:

```
AAAAAAAAAAAAAAAAB CDDDDD CDDDDD ?
```

The format of the continuation line is:

```
CDDDDD CDDDDD ?
```

- Field A is up to 16 characters of the symbol name;
- Field B is a plus (+) if the symbol name is longer than 16 characters;
- Field C is a space if the number represents a line which uses the symbol and is an asterisk (*) if the number represents a line which defines the symbol; and
- Field D is the line number which either defines or uses the symbol.

Directives

Assembler *directives* are commands, also known as *pseudo−ops*, that tell the assembler information about a program rather than directly generating object code. The following chart describes the metanotation used in the directive definitions.

expr
: any arbitrary expression

abs_expr
: an absolute expression

restricted_expr
: may contain absolute expressions and relocatable expressions consisting only of constants and symbols previously defined within a section in the current input file

symbol
: a symbol name

size
: one of **b**, **w**, or **l**, indicating an operand size of 8−, 16− or 32−bits.

fsize

one of **b**, **w**, **l**, **s**, **d**, or **x**, indicating an integer operand size of 8−, 16−, or 32−bits, or a floating−point operand type of IEEE single precision, IEEE double precision, or Motorola 68881 extended precision

flag

directive−specific modifier

string

an arbitrary series of characters, sometimes delimited with quote characters

Non−debugging directives

comm

Define a common symbol.

dc

Define constants.

ds

Define storage.

else

Conditional else.

end

End of source.

endif

End of conditional.

endm

End of macro definition.

equ

Assign a value to a symbol.

err

Print a user−defined error message.

exitm

Exit a macro.

export

Export a symbol.

fcb

Define constants.

fcc

Define constants.

fdb

Define constants.

file

Set the source file name for debugging.

if

Start of conditional assembly.

ifeq

Start of conditional assembly.

ifne

Start of conditional assembly.

ifge

Start of conditional assembly.

ifle

Start of conditional assembly.

ifgt

Start of conditional assembly.

iflt

Start of conditional assembly.

ifn

Start of conditional assembly.

ifc

Start of conditional assembly.

ifnc

Start of conditional assembly.

include

Read a source file.

interrupt

Set the interrupt flag of a symbol.

import

Import a symbol.

macro

Define a macro.

lib

Read a source file.

list

Turn on assembly listing.

llen

Set the maximum length of an assembly listing line.

nolist

Turn off assembly listing.

offset

Define offset symbols.

opt

Set assembler options.

org

Set an origin.

page

Go to the next page in the assembly listing.

plen

Set the page length of the assembly listing.

protected

Set the protected flag of a symbol.

repeat

Repeat a line a number of times.

rmb

Reserve memory bytes.

section

Define a named section.

set

Assign a value to a symbol.

special

Set the special flag of a symbol.

sttl

Set an assembly listing subtitle.

title

Set an assembly listing title.

xdef
: Export a symbol.

xref
: Import a symbol.

Debugging directives

Debugging directives are typically used in assembly files generated by high level language compilers to transmit information about the source program, such as type information and scope rules, to a source level debugger.

Any debugging directive that references another type can do so with the name of the type, or, if the type is anonymous, an asterisk (`*') can be used to indicate the type is defined in a subsequent line. If there is more than one asterisk argument to a debugging directive, the types will be defined in subsequent lines in the order they appear in the argument list. If an asterisk argument contains asterisk arguments, all type definitions will be satisfied before the next argument of the first type.

Many of the debugging directives use *atomic* data types as arguments. These data types are identified by small, positive integers. The predefined types are:

0
: undefined/void data type

1
: signed 8−bit character

2
: unsigned 8−bit character

3
: signed 8−bit integer value

4
: signed 16−bit integer value

5
: signed 32−bit integer value

6
: unsigned 8−bit integer value

7
: unsigned 16−bit integer value

8
: unsigned 32−bit integer value

9
: single precision IEEE floating point value (32−bit)

10
: Double precision IEEE floating point value (64−bit)

Type directives

array
: Define an array type.

basic
: Define a basic type.

carray

Define a C array type.

enum
: Define an enumerated type.

func
: Define a function type.

pointer
: Define a pointer type.

record
: Define a record type.

setof
: Define a set type.

struct
: Define a struct type.

subr
: Define a subrange type.

tag
: Define a structure tag.

type
: Define a type alias.

union
: Define a union type.

Special symbol directives

field
: Define a bit field.

literal
: Define an enumeration literal.

local
: Define a local variable.

member
: Define a struct or union member.

param
: Define a function parameter.

Marker directives

bbegin
: Mark the beginning of a block.

bend
: Mark the end of a block.

fbegin
: Mark the beginning of a function.

fend
: Mark the end of a function.

fentry
: Mark the entry point of a function.

fexit
: Mark the exit point(s) of a function.

label

Define a program label.

line

Define a program line number.

array

Define an *array* type.

```
[type]          array   size,index_type,element_type
```

type
: Optional name of array type

size
: An absolute expression giving the total size in bytes including any alignment

index_type
: Type of the array index, specified with either a type name of asterisk (*)

element_type
: Type of the elements, specified with either a type name or an asterisk (*)

basic

Define a *basic* type.

```
[type]          basic           number
```

type
: Optional name of the type

number
: An absolute expression representing an atomic data type (see list of predefined atomic data types)

bbegin

The bbegin directive is used to mark the beginning of a block. The following scoping rules apply to symbols and symbol references inside of a block:

- Any symbols declared inside of a block will only be accessible inside that block and any blocks declared inside of it.
- Symbols declared inside of a block override symbols of the same name declared outside the block.
- Symbols declared outside of a block are accessible, provided their names are different than symbols declared inside the block.
- Both symbols declared inside of a block and enclosing blocks can be forward referenced from inside the block.
- Symbols declared inside of inner blocks are not accessible in an outer block.
- No symbol declared inside a block may be exported
- Only debugging symbols are affected by **bbegin** and **bend** scoping.
- All **bbegin** directives must be balanced with a **bend** directive to end the block.

bend

The **bend** directive marks the end of a block declared with a **bbegin** directive.

carray

Define an *C array* type.

```
[type]          carray          size,num,element_type
```

type
: Optional name of the array type

size
: An absolute expression giving the size of the array in bytes including any alignment

num
: An absolute expression giving a positive number of array elements

element_type
: The type of the elements, specified with either a type name or an asterisk

comm

symbol comm ***abs_expr***

Define a global symbol with a size of ***abs_expr***. The linker will allocate ***abs_expr*** bytes in the first section flagged with **comms** that can hold it. (See the section directive.)

If the symbol is redefined, the largest ***abs_expr*** is used by the linker.

Given an assembler source file named *x.s* containing:

```
baz             comm            10
```

an assembler source file named *y.s* containing:

```
baz             comm            20
```

and a linker command file named *xy.ld* containing:

```
section         .xy comms origin 0x1234;
```

The result of linking would be an object file containing a single 20 byte section named *.xy*. The global symbol *baz* would have an address of $1234.

dc

[***symbol***] dc[.***fsize***] ***expr***{,***expr***}

Generate one or more constants in the current section. The size of the constant is defined by the ***fsize*** modifier, with **b** (byte) the default. If ***expr*** is a string deliminated with single or double quote characters, a constant is generated for each character in the string of that character's ASCII value. A C compatable, null terminated string can be defined:

```
hello   dc.b    'hello world',$0a,0
```

ds

`[`***`symbol`***`]` `ds[.`***`size`***`]` ***`abs_expr`***

Allocate ***abs_expr*** of ***size*** in the current section at the current location. If ***size*** is ommited, **b** (byte) is assumed. If the current section is a **bss** section, the allocated space will be uninitialized. In other types of sections the allocated space will be zero filled, see Sections.

If ***abs_expr*** is equal to zero, the assembler will add anough space to align the following code or data to the next even ***size*** address. A size of **b** will cause no space to be added, a size of **w** will align to an even byte address, and a size of **l** will align to the next address exactly divisible by four.

else

```
else
```

This reverses the effect of the most recent **if**, **ifc**, **ifn**, or **ifnc** conditional assembly directive as follows: if the statements preceding the **else** directive were being assembled because of an **if**−type conditional assembly directive, then any code following the **else** will be skipped until a terminating **endif** directive is encountered. Conversely, if the code preceding the **else** was being skipped because of a previous **if**−type directive, than any code following the **else** directive will be assembled until an **endif** occurs.

end

```
end
```

Defines the end of the assembly input. Subsequent input is ignored.

endif

```
endif
```

Marks the end of a block of conditional assembly.

endm

```
endm
```

Marks the end of a macro definition.

enum

Define an *enumerated* type.

```
[type]          enum          number
```

type
: Optional name of the enumeration type

number
: An absolute expression representing the atomic data type required to contain the enumeration (see list of predefined atomic data types)

equ

symbol equ ***expr***

Assigns the value of ***expr*** to ***symbol***.

err

```
err              {string}
```

Cause the assembler to generate a user−defined error message. If ***string*** is not specified, the message `user−specified error' is generated.

exitm

```
exitm
```

This directs the assembler to skip to the next **endm** directive, leaving the current macro.

export

```
export[.w]          symbol{,symbol}
```

The **export** directive specifies that the comma separated list of ***symbols*** is defined within the current source file, but should be passed to the linker so the other programs may reference them. An exported symbol can be referenced in another file using the **import** directive. All symbols listed must be defined within the current source file. **export** modified with **.w** exports the symbols *weakly*: they will not be entered in an archive symbol table and will only resolve imports if some other symbol in the library module is both referenced and exported normally. A colon following a symbol where it is defined implicitly exports the symbol, two colons cause the symbol to be exported weakly. This is a synonym for **xdef**.

fbegin

fbegin type
...
endl

The **fbegin** directive marks the beginning of a function. All **fbegin** directives must be balanced with an **fend** directive. All symbols and symbol references defined inside of a function follow the same rules as a block defined with the **bbegin** and **bend** directives, except that all symbols, except those that begin with a question mark, obey scoping rules.

The second form of the **fbegin** directive causes the assembler to issue an implicit **func** directive with *type* as its argument.

fcb

[***symbol***] fcb ***expr***{,***expr***}

Generate one or more byte constants in the current section.

fcc

```
[symbol]                fcc                     expr,string
[symbol]                fcc                     <delim>string<delim>
```

Generate a string constant in the current section. There are two forms of this directive. In the first form the length of the string is given with ***expr*** and the contents of the string are given with ***string***. If string is longer than expr bytes, it is truncated. If ***string*** is shorter than ***expr*** bytes, the ***string*** is padded on the end with space characters (ASCII $20). ***expr*** must start with a numeric digit (0−9).

In second form of the **fcc** directive the first non−blank, non−numeric character is the string deliminator. All characters in string until the deliminator character is encountered will be included.

```
hello   fcc    'hello world'
goodbye fcc    10,mystring              ; this will be padded to 10 bytes
```

fdb

[***symbol***] fdb ***expr***{,***expr***}

Generate one or more double byte constants in the current section.

fend

The **fend** directive marks the end of a block declared with an **fbegin** directive.

fentry

The **fentry** directive is used to mark the *entry point* of a function or procedure. It is always used inside of a function defined with **fbegin** and **fend** directives.

fexit

The **fexit** directive is used to mark the *exit point(s)* of a function or procedure. It is always used inside of a function defined with **fbegin** and **fend** directives.

field

Define a *bit field*.

name `field` ***displacement*,*bits*,*host***

name
: Name of the bit field

displacement
: An absolute expression giving the number of bits the first bit of the bit field is from the beginning of the structure containing the bit field

bits
: An absolute expression giving the number of bits in the field

host
: The type name of the host type, specified with either a type name or an asterisk (*)

file

```
file            string
```

Inserts a ***string*** into the file record field of the ICOFF file, used by debuggers to locate the corresponding source file.

func

Define a *function* type. Zero or more parameter definition directives may follow the **func** directive: the list is terminated with an **endl** directive.

```
[type]          func            return_type
pname           param           offset,type
                ...
                endl
```

type

Optional name for the function type

return_type

The name of the return type, specified with either a type name or an asterisk (*). If the function has no return value, the return type is specified with a period.

if

```
if              abs_expr
```

If the ***abs_expr*** evaluates non−zero, assemble the code following until an **else** or **endif** directive occurs. See Conditional assembly.

ifc

```
ifc             string,string
```

If the two ***strings*** are identical, the code immediately following the **ifc** directive line will be assembled until either an **else** or **endif** directive occurs. Quote marks (") are optional on both sides of the string. If the quotes are left off, the first string ends at the comma, and the second string ends at the first whitespace character after the comma.

ifeq

```
ifeq            abs_expr
```

If the ***abs_expr*** is equal to zero, assemble the code following until an **else** or **endif** directive occurs.

ifge

```
ifge            abs_expr
```

If the ***abs_expr*** is greater than or equal to zero, assemble the code following until an **else** or **endif** directive occurs.

ifgt

```
ifgt            abs_expr
```

If the ***abs_expr*** is greater than zero, assemble the code following until an **else** or **endif** directive occurs.

ifle

```
ifle            abs_expr
```

If the ***abs_expr*** is less than or equal to zero, assemble the code following until an **else** or **endif** directive occurs.

iflt

```
iflt            abs_expr
```

If the ***abs_expr*** is less than zero, assemble the code following until an **else** or **endif** directive occurs.

ifn

```
ifn             abs_expr
```

If the ***abs_expr*** evaluates to zero, assemble the code following until an **else** or **endif** directive occurs.

ifnc

```
ifnc            string,string
```

If the two strings are different, the code immediately following the **ifnc** directive line will be assembled until either an **else** or **endif** directive occurs. Quote marks (") are optional on both sides of the string. If the quotes are left off, the first string ends at the comma, and the second string ends at the first whitespace character after the comma.

ifne

```
ifne            abs_expr
```

If the ***abs_expr*** is not equal to zero, assemble the code following until an **else** or **endif** directive occurs.

import

```
import[.s]          symbol{,symbol}
```

The **import** directive specifies that the comma separated list of ***symbols*** which are defined in a separate source file (or files) should be accessible in the current source file. This symbols must be exported from the other source file (or files), or linking will fail. The optional **.s** after the **import** directive causes the symbol to be referenced using base page addressing in **as05**, **as08**, **as09**, **as11**, and **as12** and short absolute addressing in **as68**. This is a synonym for **xref**.

include

```
include              string
```

The **include** directive tells the assembler to read in the file named ***string*** and insert it in place of the **include** directive in the assembly code. The ***string*** can be delimited with either single quotes (') or double quotes ("). The delimitiers are required only for file names that contain spaces.

The default directories that are searched for files are (in order):

- The current directory
- Directories specified with the **−y** option
- $INTROL/Libraries/C
- $INTROL/Libraries/C/lib*XX*
- $INTROL/Libraries/Assembly
- $INTROL/Libraries/Assembly/gen*XX*

where *XX* is the two digit assembler suffix.

The −y and −yy command line options specify where, besides the default directories, files should be found. **include** is a synonym for **lib**.

interrupt

```
interrupt          symbol{,symbol}
```

The **interrupt** directive specifies that the comma separated list of ***symbols*** should be marked with the **SF_INT** symbol flag in the ICOFF output file. This flag is used during **icode** code generation.

label

Define a label.

```
label          name
```

name
Name of the label

lib

```
lib             string
```

The **lib** directive tells the assembler to read in the file named ***string*** and insert it in place of the **lib** directive in the assembly code. The ***string*** can be delimited with either single quotes (') or double quotes ("). The delimitiers are required only for file names that contain spaces.

The default directories that are searched for files are (in order):

- The current directory
- Directories specified with the **−y** option
- $INTROL/Libraries/C
- $INTROL/Libraries/C/lib*XX*
- $INTROL/Libraries/Assembly
- $INTROL/Libraries/Assembly/gen*XX*

where *XX* is the two digit assembler suffix.

The −y and −yy command line options specify where, besides the default directories, files should be found. **lib** is a synonym for **include**.

line

Create a *line number* record.

```
line            number
```

number
An absolute expression. This creates a line number entry in the ICOFF file equal to number and the current offset in the current section.

list

```
list
```

The **list** and **nolist** directives implement a nestable listing control system. This is unlike **opt l** and **opt nol**, which completely turn the listing on and off respectively.

The assembler maintains a list level which starts out at zero. Whenever the list level is zero, the listing is enabled. The **nolist** directive increments the list level and the **list** directive decrements the list level.

literal

Define a *literal*.

```
[name]          literal        value
```

name
: Optional name of the literal

value
: An absolute expression giving the actual value of the literal

llen

```
llen            abs_expr
```

Sets the default line length in the listing file for the symbol table, the cross reference and the page headers. It does not affect the source code or section synopsis listing.

local

Define a *local variable*.

name `local` ***offset*****,*****type*****{,reg}**

name

Name of the local variable

offset

An absolute expression giving a machine dependent offset of the local variable from a stack or frame pointer, or the machine dependent register number containing the local variable. The offset can be the name of a processor register, in which case the **reg** flag will be set implicitly.

type

The local variable's type, specified with either a type name or an asterisk (*)

reg

An optional flag that can be specified to define a register variable

macro

```
symbol          macro           {symbol{,symbol}}
                ...
                endm
```

Defines a macro containing any assembly source. The **endm** directive signals the end of the macro. The ***symbols*** are dummy parameter names that are expanded within the macro body.

member

Define a structure or union *member*.

name member ***offset,type***

name
: Name of the member

offset
: An absolute expression giving the byte offset of the member

type
: The member's type, specified with a type name or an asterisk (*)

nolist

```
nolist
```

Disables listing until directed otherwise.

offset

```
offset          abs_expr
```

offset creates a section−like environment within the assembly source file that is used to create absolute symbols.

opt

```
opt              string{,string}
```

where string is any of the following specifiers:

cl
: Print conditional assembly directive lines

cpu_68hc11
: **as11 only.** Set processor type to 68HC11

cpu_630x
: **as11 only.** Set processor type to 6301, 6303 (Hitachi)

cpu_680x
: **as11 only.** Set processor type to 6801, 6803

cpu_68000
: **as68 only.** Set processor type to 68000

cpu_683xx
: **as68 only.** Set processor type to the 68300 family

cpu_68010
: **as68 only.** Set processor type to 68010

cpu_68020
: **as68 only.** Set processor type to 68020

cpu_68030
: **as68 only.** Set processor type to 68030

cpu_68ec030
: **as68 only.** Set processor type to 68ec030

cpu_68040
: **as68 only.** Set processor type to 68040

cpu_68ec040
: **as68 only.** Set processor type to 68040

fpu_68881
: **as68 only.** Set floating−point processor type to 68881/2

fpu_68040
: **as68 only.** Set floating−point processor type to 68040

frl
: **as68 only.** Assume forward references to be long, when the size can vary

frs
: **as68 only.** Assume forward references to be short

g
: Print data lines for data definitions

inc
: Print include and li lib.html b files in listing.

l
: Print source listing from this point (default)

mc
: Print macro calls (default)

md
: Print macro definitions (default)

mex
: Print lines generated by macro expansions (default)

ov

as68 only. Convert add to addq, and move to moveq when possible

pcl

as68 only. Assume pc−relative offsets to be long, when the size can vary

pcs

as68 only. Same as nopcl

If an option is prefaced by `no', the sense of the option is negated.

org

```
org             restricted_expr
```

In this context, ***restricted_expr*** may contain absolute expressions and relocatable expressions consisting only of constants and symbols previously defined within a section in the current input file. If the expression is absolute, the assembler creates an absolute section at the address specified by ***restricted_expr***. If there was a previous **org** directive with the same address, the assembler overwrites the section at that address. If the expression is relocatable, the assembler sets the current location to ***restricted_expr*** in the section containing the definition of the relocatable symbol used in ***restricted_expr***.

page

```
page
```

Creates a new page in the listing file.

param

Define a *parameter* for a function.

```
[name]          param          offset,type[,reg][,alias]
```

name
: Optional name of the parameter

offset
: An absolute expression giving a machine dependent byte offset from a stack or frame pointer, or an absolute expression giving a machine dependent register number. Note: the register number is only used in conjunction with the **reg** flag. The offset can be the name of a processor register, in which case the **reg** flag will be set implicitly.

type
: The parameter's type, specified with either a type name or asterisk (*)

reg
: An optional flag used to define a register parameter

alias
: An optional flag used to define alias parameters used for non−local variable referencing

plen

```
plen            abs_expr
```

Set the length of the page used in the listing file.

pointer

Define a *pointer*.

`[`***type***`]` `pointer` ***num*,*ptype***

type
: Optional name for the pointer type

num
: An absolute expression giving the atomic type of the pointer (see list of predefined atomic data types).

ptype
: The type pointed to by the pointer, specified with either a type name or an asterisk (*).

protected

```
protected          symbol{,symbol}
```

The **protected** directive specifies that the comma separated list of ***symbols*** should be marked with the **SF_PROT** symbol flag in the ICOFF output file. This flag is used during **icode** code generation.

record

Define a *record*. Zero or more members may follow the record directive. The list is terminated with an endl directive.

```
[type]          record          size
name            member          offset,mtype
                ...
                endl
```

type

Optional name for the record type

size

An absolute expression giving the size in bytes of the record including any alignment

repeat

```
repeat          abs_expr
repeat          abs_expr,abs_expr
```

The first form causes the next line in the assembly file to be repeated a number of times indicated by ***abs_expr***. The second form causes the next line in the assembly file to be repeated a number of times indicated by the absolute value of the difference between the two expressions. In addition, the initial value of the \r counter is set to the first argument.

rmb

{***symbol***} rmb ***abs_expr***

Allocate ***abs_expr*** bytes in the current section at the current location. **rmb** is a synonym for **ds.b**.

section

```
section         symbol[{,addr=abs_expr[{,flag}]}]
section.s       symbol[{,addr=abs_expr[{,flag}]}]
```

section will direct the assembler to create or switch to the section named ***symbol***. If an address is specified by ***abs_expr***, the section is made absolute. ***flag*** can be one of the following:

abs
: The section is absolute. **addr=*abs_expr*** should be specified.

bss
: The section is uninitialized.

comm
: The section is a common section.

data
: The section contains program data.

text
: The section contains program executable code.

icode
: The section contains ICODE commands.

reg
dsect
noload
: These section flags are not used by the Introl−CODE system.

The **section.s** form causes all symbols defined in the section to be accessed using absolute short addressing (for the 68000 family) or base page addressing (other processors) where appropriate.

set

symbol `set` ***restricted_expr***

Assigns the value of ***restricted_expr*** to ***symbol***. In this context, ***restricted_expr*** may contain absolute expressions and relocatable expressions consisting only of constants and symbols previously defined within a section in the current input file. Only addition and subtraction are allowed in relocatable expressions. Symbols defined using **set** may be redefined elsewhere in the program with another **set** directive. The symbol may not be made global or exported.

setof

Define a *set*.

```
[type]          setof          num,subtype
```

type
: Optional name of the set type

num
: An absolute expression giving the atomic data type required to represent the set (see list of predefined atomic data types).

subtype
: The name of a type representing the range of the set, specified with either a type name or an asterisk (*).

special

```
special          symbol{,symbol}
```

The **special** directive specifies that the comma separated list of ***symbols*** should be marked with the **SF_SPEC** symbol flag in the ICOFF output file. This flag is used during **icode** code generation.

struct

Define a *structure*. The struct's members follow the **struct** directive in order. There may be zero or more members. The list of members is terminated with an **endl** directive.

```
[type]          struct          size
name            member          offset,mtype
                ...
                endl
```

type

Optional name of the struct type

size

An absolute expression giving the size in bytes of the struct including any alignment

sttl

```
sttl            string
```

Set the subtitle to be printed on subsequent pages in the listing file. If all the lines in the source file above the **sttl** directive are operations that are normally not listed, the subtitle will take effect on the current page.

subr

Define a *subrange*.

```
[type]          subr          num,stype,from,to
```

type
: Optional name of the subrange type

num
: An absolute expression representing the atomic data type required to represent the subrange (see list of predefined atomic data types)

stype
: The type of the set members, specified with either a type name or an asterisk (*)

from
: An absolute expression giving the low value of the subrange

to
: An absolute expression giving the high value of the subrange

tag

Define a *tag*.

name `tag` ***type***

name
: Name of the tag

type
: The type represented by the tag name, specified with either a type name or an asterisk (*)

title

```
title          string
```

Set the title to be printed on subsequent pages in the listing file. If all the lines in the source file above the **title** directive are operations that are normally not listed, the title will take effect on the current page.

type

Define a *type alias*.

```
[name]          type            ttype
```

name
: Optional name of the type

ttype
: The type represented by the type name, specified by a type name or an asterisk (*)

union

Define a *union*. The union's members follow the union directive in order. There may be zero or more members. The list of members is terminated by the **endl** directive.

```
[type]          union           size
name            member          offset,mtype
                ...
                endl
```

type

Optional name of the union type

size

An absolute expression giving the size in bytes of the union including any alignment

xdef

```
xdef          symbol{,symbol}
```

The **xdef** directive specifies that the comma separated list of ***symbols*** is defined within the current source file, but should be passed to the linker so the other programs may reference them. An exported symbol can be referenced in another file using the import directive. All symbols listed must be defined within the current source file. **xdef** modified with **.w** exports the symbols *weakly*: they will not be entered in an archive symbol table and will only resolve imports if some other symbol in the library module is both referenced and exported normally. A colon following a symbol where it is defined implicitly exports the symbol, two colons cause the symbol to be exported weakly. This is a synonym for **export**.

xref

```
xref[.s]          symbol{,symbol}
```

The **xref** directive specifies that the comma separated list of ***symbols*** which are defined in a separate source file (or files) should be accessible in the current source file. This symbols must be exported from the other source file (or files), or linking will fail. The optional **.s** after the **xref** directive causes the symbol to be referenced using base page addressing in **as05**, **as08**, **as09**, **as11**, and **as12** and short absolute addressing in **as68**. This is a synonym for **import**.

The C Compilers

This chapter instructs on using the C compilers supplied with Introl−CODE. The included C compilers (and the processors they support , the default is in **bold**) are:

C compiler	Supported Processor(s) (default is **bold**)
icc	The general compiler
cc09	**6809**
cc11	**68HC11**, 6801, 6301
cc12	**68HC12**
cc16	**68HC16**
cc68	**683XX**, 68000, 68010, 68020, 68030, 68EC030, 68040, 68EC040

C compiler command line
: How to run the compilers.

Implementation
: Data type sizes and other Introl specific information.

Preprocessor defines
: Pre−defined preprocessor macros.

Include files
: How to use include files and include file search order.

Output files
: The error listing, object, assembly, and assembly listing files the compiler can produce.

Option files
: How to create a file containg a default set of options.

Debugging information
: How to compiler a program for use with a source level debugger.

Warnings and errors
: Warning and error messages produced by the compilers: what they mean and how to correct the errors.

C compiler command line

Compiler commands lines consist of the compiler name, zero or more compiler options, and the name of a source file to compile.

```
cc09 [{options}] filename

cc11 [{options}] filename

cc12 [{options}] filename

cc16 [{options}] filename

cc68 [{options}] filename

icc −gprocessor [{options}] filename
```

The various processor specific compilers, **cc09**, **cc11**, **cc12**, **cc16**, and **cc68**, are synonyms for the **icc** command with the appropriate ***−gprocessor*** selection option.

By convention, the name of C language files have the extension **.c**.

Output files produced by the C compiler are given names based on the original file name unless overridden by command line options below. The compiler will replace the source file's extension with an extension appropriate to the output file type. For example, if the original source file is *test.c*, the listing file, if any, will be placed in the file *test.lst*, and the object file will be placed in the file *test.o12* (for **cc12**).

−a[f]S=string

The **−a** option is used to override the compiler's default section usage. *S* specifies the default section to be overridden and can be one of the following characters:

t
: Override the **.text** (program executable code) section.

d
: Override the **.data** (program initialized data) section.

b
: Override the **.bss** (program uninitialized data) section.

s
: Override the **.strings** (program strings) section.

1
: Override the **.mod1** (__mod1__) extra section.

2
: Override the **.mod2** (__mod2__) extra section.

c
: Override the **.const** (program constant data) section.

p
: Override the **.base** (program absolute short or base page data) section.

If the **f** is present, and the processor supports memory models, then the appropriate far section will be redefined, see Section allocation.

The appropriate default section name will be replaced by *string* by the code generator.

For example,

```
cc11 −at=.mytext test.c
```

Will compile the file **test.c** and place any executable code in the section **.mytext**, rather than the default **.text**.

−b[=filename]
Only run the preprocessor on the source file. The preprocessor output is written to standard output or to the specified *filename*.

−c
Process the default option file immediately when the **−c** option is parsed. Normally it is read after all command line options have been parsed.

−c=filename
Use the file *filename* as an option file in addition to the default option file.

−cn
Do not read the default option file.

−C
Make string literals constant.

−Dname[=string]
Define a preprocessor macro *name* with the value `1' or *string*, if present.

−E[=filename]
Only run the preprocessor on the source file. The preprocessor output is written to standard output or to the specified *filename*.

−e[c[number]][=filename]
Generate a source listing with errors or warnings placed in context. If no **c** is specified, the entire source file will be listed. If **−ec** is specified, five lines of code above and below the line creating the error will be retained. If **−ec***number* is specified, *number* lines will be retained above and below. If a *filename* is specified, the listing will be placed in the specified file.

−g
Generate symbolic debugging information for source−level debugging.

−gprocessor
Generate code for the processor specified by *processor*. The choice of processor controls the code generator and assembler that is used by the compiler. The legal values for *processor* are given in the following table:

−g01	6801, 6803
−g03	6301, 6303 (Hitachi)
−g09 (default for cc09)	6809
−g11 (default for cc11)	68HC11
−g12 (default for cc12)	68HC12
−g16 (default for cc16)	68HC16
−g68, −g62 (default for cc68)	683XX family
−g00	68000
−g10	68010
−g20	68020/68030
−g40	68040

Without a −g*processor* option, the compiler uses the last two digits of the command by which it was invoked as a default value of *processor*.

−ganumber{c|s|l|f|d}

Set the *byte alignment* of variables in a structure or union, where number is 1, 2, or 4 (for byte−, word−, or longword alignment). The corresponding variable types are specified by: **c** for char, **s** for short, **l** for long, **f** for float, and **d** for double. Use a separate **−ga** for each type.

−gbl|h

Set the *bit field fill order* (**−gbl** is the default). **−gbl** will fill bit fields from low order to high order, while **−gbh** will fill bit fields from high order to low order.

−gcs|u

Set **char** to signed (**s**), or unsigned (**u**). By default **char** is unsigned for the 6809, 68HC11, 6801, 6301, and 68HC12, and signed for the 68HC16 and 68XXX family.

−gds|l

Set **double** to be 32−bits (**s**), or 64−bits (**l**). By default, **double** is 32−bits for the 6809, 68HC11, 6801, 6301, and 68HC12, and 64−bits for the 69HC16 and 68XXX family. 64−bit floating point data can always be defined by declaring a variable **long double**. Double−precision (64−bit) floating point routines are included only in the libraries for the 68HC16 and 68XXX family. If you specify 64−bit floating point data for a processor that does not have library support for it you will get unresolved references at link time unless you supply your own 64−bit floating point routines.

−gg

Generate symbolic debugging information for source−level debugging.

−ggn

68XXX family only. Disable *deferred stores* from being generated by the code generator.

−ggr

Generate a **nop** after every **rts**. This option is used to work around a problem in some hardware emulators that an execution breakpoint on the byte after an executed **rts** would get triggered.

−gh

68HC16 only. Use function calls to load and store **double** values instead of generating in−line code. This saves space at the expense of speed.

−gis|l

68HC16 and 68XXX family only. Set the size of **int** to 16−bits (**s**) or 32−bits (**l**). The default size of **int** is 16−bits for the 68HC16 and and 32−bits for the 68XXX family. Use of this option with other than the default value will make your program incompatible with the Introl supplied libraries.

−gk

68HC16 only. Disable the assumption that the stack resides in the near bank. See Memory model related extensions.

−gl

68HC11, 68HC12: Set compiler to *function large model*. The compiler behaves as if a **__far** type modifier has been applied to all functions and function pointers.
68HC16 only. Set compiler to *large model*. The compiler behaves as if a **__far** type modifier has been applied in every context where an explicit **__near** or **__far** modifier has not been used. See Memory model related extensions.

−gm[number]

68XXX family only. Generate code for the math co−processor (if supported).

number	Co−processor supported
none	Generate floating−point instructions for the 68881/68882.
1	Generate floating−point instructions for the 68040 and also generate the **fintrz** instruction.

2	Synonymous with −gm.
5	Generate floating−point instructions for the 68040. Use in−line code for rounding to an integer.
9	Generate floating−point instructions for the 68040. Use a library function for rounding to an integer.

−go

Generate icode intermediate code rather than object code.

−grc|d|C

6809 and 68XXX family only. The compiler will generate position−independent (**c**)ode, (**d**)ata, and/or (**C**)onstants, if supported by the processor and it's code generator.

−gs

6809 only. Create a call stack frame using the **u** index register.

−gssize

68HC16 only. The 68HC16 compiler doesn't always deallocate stack memory after a function call. At times several function calls may occur before the parameter stack is deallocated. The **−gs** option sets the number of bytes of stack growth should be allowed before parameters are deallocated. If the *size* argument is negative, then parameters will be deallocated after each function call.

−gv[number]

68HC11 and 68HC12 only. Change the compiler's assumed bytes−per−expression from its default of 5 to *number*. This value is used by the compiler to avoid generating **brset** or **brclr** instructions that could not reach their targets. It is used as a measure of code density. If you get a **brset** or **brclr** branch out of range error, raising this value will eliminate it. If no *number* is specified, **brset** and **brclr** instructions are not generated.

−gv[number]

68HC16 only. Sets the size, in 16 bit words, at which a function call is used for copying a **struct**, rather than in−line code. The default value is 6.

−gw

68HC11 only. Disable generation of the **brclr** instruction for tests of an 8−bit value against 0.

−gwnumber

68HC12 and 68HC16 only. Determines which kind of switch table to generate. The compiler computes the code size for a faster table lookup method and a slower binary search method. A table lookup method is preferred unless it would generate *number* more bytes than the binary search method. The default *number* is 100 bytes.

−gx

68HC11 only. Put switch table data in the **.const** section instead of **.text**.

−gx

68XXX family only. Ignore the **register** keyword. The compiler normally gives greater weight to variables defined as **register** when allocating register variables.

−gy

Mimic the calling conventions of the Version 3.06 compiler.

−Idirectory

Add a *directory* to the compiler's header file search path.

−i[=]directory

−ii[=]directory

Add a *directory* to the compiler's header file search path. If **−i** is specified, the current directory is searched first; if **−ii** is specified, the current directory is searched last.

−k

Print the name and version number of each compilation pass.

−l[=filename]
Create an assembly listing file while compiling. *filename*, if present, specifies the listing file name. By default, the assembler creates a listing file whose name is based on the C source file name but has an extension of **.lst**.

−L
Allow the result of a cast to be an lvalue (backward compatability option).

−mname[=string]
Define a preprocessor macro *name* with the value `1' or *string*, if present.

−o=filename
Set the output object file name to be *filename*. If no extension is given on *filename*, then an extension of **.o*XX***, where *XX* is the processor number, will be used.

−P
Read standard input instead of *filename* for the program source. *filename* is still required to allow the compiler to create appropriately named temporary and object files.

−r
Retain the generated assembly language file after the assembler has completed. The default is to delete the assembly file once the object file has been created.

−S
Retain the generated assembly language file after the assembler has completed. The default is to delete the assembly file once the object file has been created.

−s
With this option enabled, the compiler will not be strictly ANSI compatible. This is one of the compatibility options for Version 3.06. This option also implies **−gy**. **Caution**: The −s option may require recompilation of the libraries to match the old calling convention.

−t[=]directory
Set the name of the *directory* used for temporary files.

−Ustring
−ustring
Undefine the preprocessor macro *string*.

−w[number]
Set the warning level to *number* between zero and nine. If the **−w** option is used without a *number*, the warning level is set to zero. The default warning level is five. **−w9** displays all warnings.

−we
Turn all compiler warnings into errors.

−x
Check for syntax and semantic errors only, don't create an object file.

−X
Compile the source file to assembly language but don't assemble it.

−y[=number]
Set the maximum length for an identifier. If no *number* is specified, the length is set to eight. The default is 90.

Implementation

Types, pointers and conversions

Types are divided into two main classes: *fundamental types* and *derived types*. You can construct an infinite number of derived types from combinations of fundamental types or already defined derived types.

Fundamental data types

The supported fundamental types and their sizes and attributes are listed in the table below.

Fundamental types for the 6809, 68HC11, and 68HC12.

Type	Size in bits	Signed range	Unsigned range
char	8	−128 .. 127	0 .. 255
short	16	−32768 .. 32767	0 .. 65535
int	16	−32768 .. 32767	0 .. 65535
long	32	−2147483648 .. 2147483647	0 .. 4294967296
float	32	±1.2x10^−38 .. ±3.4x10^38	
double	32	±1.2x10^−38 .. ±3.4x10^38	
long double *****	64	±2.2x10^−308 .. ±1.8x10^307	

Fundamental types for the 68HC16.

Type	Size in bits	Signed range	Unsigned range
char	8	−128 .. 127	0 .. 255
short	16	−32768 .. 32767	0 .. 65535
int	16	−32768 .. 32767	0 .. 65535
long	32	−2147483648 .. 2147483647	0 .. 4294967296
float	32	±1.2x10^−38 .. ±3.4x10^38	
double	64	±2.2x10^−308 .. ±1.8x10^307	
long double	64	±2.2x10^−308 .. ±1.8x10^307	

Fundamental types for the 68XXX Family.

Type	Size in bits	Signed range	Unsigned range
char	8	−128 .. 127	0 .. 255
short	16	−32768 .. 32767	0 .. 65535
int	32	−2147483648 .. 2147483647	0 .. 4294967296
long	32	−2147483648 .. 2147483647	0 .. 4294967296
float	32	±1.2x10^−38 .. ±3.4x10^38	
double	64	±2.2x10^−308 .. ±1.8x10^307	
long double	64	±2.2x10^−308 .. ±1.8x10^307	

***** Although **long double** is supported by the 6809, 68HC11, and 68HC12 C compilers, there is no support for long doubles in the runtime support libraries for these processors.

char, signed char and unsigned char

A **char** is defined to be large enough to store any character from the machine's character set (assumed to be ASCII) as a positive number. All character variables are implemented as 8−bit bytes. Introl−C supports both signed and unsigned characters. In the absence of a −gc option, the plain **char** type is signed for the 68HC16 and 68XXX family and unsigned for the 6809, 68HC11, 6801, 6301, and 68HC12.

unsigned short, unsigned long, signed long and signed short

Integers are used to represent integral quantities. Integer data objects can be declared in various sizes by use of an optional modifier. Integers come in two sizes: **short** and **long**. Introl−C compilers ensure that all **short** integers are 16−bit and **long** integers are 32−bit. Normal integers are the appropriate length for the machine in use. All signed integers are represented in 2's complement form. Unsigned integers represent positive quantities.

float, double and long double

Floating−point numbers are represented in the IEEE standard floating−point format. A float is 32−bit and a long double is 64−bit. The size of a double is either 32− or 64−bits depending on the target.

Pointers

Pointers are equivalent to **int** sized unsigned integers, with the following exceptions:

- On the 68HC11 and 68HC12, function pointers are 32−bits if the pointer is a far pointer.
- On the 68HC16, function pointers are always 32−bits and other pointers may be, depending on their declaration and the memory model.

Type conversions

Converting an integer type to an integer type of equal size does not change the bit pattern. Converting a larger integer type to a smaller integer type is performed by truncation. Converting a smaller integer type to a larger integer type is performed by sign extension if the source type is signed, or by zero extension if the source type is unsigned.

Storage classes

Register

The compiler treats the **register** keyword as a suggestion to place the variable in a register if possible. Whether or not a variable was actually placed into a register, you cannot determine its address with an ampersand operator.

Volatile

Using **volatile** keyword tells the compiler that a variable can be externally modified. This causes the compiler to reload the value of the variable every time it is used. Generally, all device registers should be declared as **volatile**.

Preprocessor defines

You can define *preprocessor macros* on the command line using the **−m** or **−D** options. For example, the following command:

```
cc09 −mABC=5 test.c
```

would have the same effect as if the compiler were to have read the directive:

```
#define ABC 5
```

in a C source file. If you do not specify an expansion for a command line defined macro, the macro will be defined as ``1". For example, the following command:

```
cc16 −DABC test.c
```

has the same effect as if the compiler were to have read the preprocessor directive:

```
#define ABC 1
```

If you want to define a macro that expands to a null string, use = with no argument:

```
cc11 −mABC= test.c
```

which has the same effect as:

```
#define ABC
```

namely, it will define *ABC* so it expands to an empty string.

Predefined macros

The Introl−C preprocessor defines the following macros:

macro	**definition**
_ _LINE_ _	current line number
_ _FILE_ _	current source file name
_ _STDC_ _	1 − indicates Level−1 ANSI compliance
_ _CC09_ _	non−zero − indicates that cc09 was executed
_ _CC11_ _	non−zero − indicates that cc11 was executed
_ _CC01_ _	non−zero − indicates that cc11 −g01 (6801) was executed
_ _CC03_ _	non−zero − indicates that cc11 −g03 (6301) was executed
_ _CC12_ _	non−zero − indicates that cc12 was executed
_ _CC16_ _	non−zero − indicates that cc16 was executed
_ _CC68_ _	non−zero − indicates that cc68 (683XX) was executed (_ _CC62_ _ will also be defined)

_ _CC00_ _	non−zero − indicates that cc68 −g00 (68000) was executed
_ _CC10__ _	non−zero − indicates that cc68 −g10 (68010) was executed
_ _CC20_ _	non−zero − indicates that cc68 −g20 (68020) was executed
_ _CC40_ _	non−zero − indicates that cc68 −g40 (68040) was executed
_ _CHAR_SIGNED_ _	1 − indicates that **char** is signed during compilation
_ _INT_SHORT_ _	1 − indicates that **int** is the same size as **short** during compilation
_ _DOUBLE_SHORT_ _	1 − indicates that **double** is the same size as **float** during compilation
_ _COP_ _	null − defined if the −gm command line option was used
_ _DATE_ _	date at compilation
_ _TIME_ _	time at compilation

In the preceding table, ``null'' means that the macro has an empty definition (it is defined as a null string). You can use the preprocessor **#ifdef** command to test for these names.

You can undefine predefined macros using the **−u** option. For example, if you use the following command:

```
cc68 −u_ _DATE_ _ test.c
```

the symbol **_ _DATE_ _** will not be defined by the preprocessor.

Include files

The compiler searches *$INTROL/include* to find files specified by an **#include** directive. To direct the compiler to search other directories, use the **−i** and **−I** options. The compiler will accept up to 50 directories specified with **−i** and will search those directories in order before *$INTROL/include*. Unless you specify otherwise, the compiler checks the current directory first. For example, to make the compiler search in the directory */usr/abc* on a Unix system:

```
cc68 −I/usr/abc test.c
```

or to search the directory *\incl* on drive *D* of an MS−DOS system:

```
cc12 −i=d:\incl test.c
```

To direct the compiler to search the current directory last, specify at least one alternate directory using the **−ii** option. The **−ii** option tells the compiler to search the current directory last. For example, given the following command:

```
cc16 −i=/usr/abc −ii=/usr/def
```

the compiler will search the following directories (in the order listed) for **#include** files:

- */usr/abc*
- */usr/def*
- *$INTROL/include*
- current directory

Contrast this with the following command:

```
cc09 −i=/usr/abc −i=/usr/def test.c
```

which would make the compiler search the directories in the following order:

- current directory
- */usr/abc*
- */usr/def*
- *$INTROL/include*

Output files

Listing files

Introl−C can produce a listing of your C source file that shows where errors or warnings occur. By default, the **−e** option causes a listing to be displayed on your screen. By default the entire source file and all error and warning messages are included in the listing. To place the listing into a file, specify it as follows:

```
cc68 −e=abc.lst abc.c
```

For example, if the file abc.c contained:

```
main()
        {
        int a, b;
        a = b;
        b = c;
        }
```

the previous compilation command would produce the file *abc.lst* containing:

```
            File:                   z.c
Line                    Source
----                    ------------------------------------------------
1:                      main()  {
2:                              int a, b;
3:                              a = b;
4:                              b = c;
******                  − undeclared identifier − c
5:                              }
******                  − symbol used but not defined in block − c
2 errors, 0 warnings detected
```

You can specify the number of source lines to include as context surrounding each error. For example, the command:

```
cc68 −ec=abc.lst abc.c
```

would create a listing file with the five lines of context before and after any error or warning message displayed.

```
cc68 −ec2=abc.lst abc.c
```

would behave as before, but only two lines of context would be included in the listing before and after each error or warning.

Object and assembly files files

The compiler normally produces *relocatable object* files (in ICOFF) from C source files. For example, the following command line:

```
cc68 test.c
```

produces the relocatable object file *test.o68* from the C source file *test.c*. If you want to retain the assembler source file that the compiler produced, use the **−r** or **−S** options. The following command:

```
cc12 −r test.c
```

produces both the object file *test.o12* as well as the assembler source file *test.s12*.

The assembler source file is always produced by the compiler, but it is automatically deleted after the assembler has finished, unless you use either the **−r** or **−S** option.

If you want to see the assembler listing file, use the **−l** option. The following command:

```
cc16 −S −l test.c
```

produces the assembler listing file *test.lst* in addition to the object file *test.o16* and the assembler source file *test.s16*. If you want the object file to be named differently, or placed in another directory, use the **−o** option. The following command:

```
cc68 −o=abc.o68 test.c
```

produces the object file *abc.o68* after compiling *test.c*.

Option files

An option file contains a list of command line options, one option per line. The −c option controls the compiler's use of option files. Comments may be introduced in an option file by an asterisk (*) or a semicolon (;). These comment characters cause the rest of the line on which they occur to be ignored. Option files are searched for in the directory specified by the environment variable **IOPTDIR** or in the current directory. Use an option file when the same set of options are routinely being used during many different compilations.

The default name of the option file is **optcc*XX***, where *XX* is the processor number used in the compiler command.

Debugging Information

The compiler normally produces relocatable object files in ICOFF with line number information, addresses of global and static symbols (all functions and variables) and only provides enough information for assembly level debugging. For example, the single source file program *test.c* compiled and linked with the following commands:

```
cc68 test.c
ild68 −gc68 test.o68 −o test
```

would produce the program *test* suitable only for assembly level debugging.

To produce complete symbol table information suitable for source level debugging, use the −g or **−gg** options. For example, if the program *test.c* were compiled and linked with the following commands:

```
cc09 −gg test.c
ild09 −gc09 test.o09 −o test
```

the file test would contain enough information for source level debugging. Notice that linking does not change when preparing a program for debugging.

Warnings and errors

The compiler produces warnings and errors when it encounters input it cannot parse, as well as when it encounters input that is known not to conform to ANSI programming standards or other accepted practices. There are four categories: preprocessor warnings and errors, and parser warnings and errors.

Preprocessor warnings

divide or mod by zero, expression evaluates to 1
: Preprocessor expression evaluation resulted in a divide or mod by zero. The expression defaults to 1 (true).

unrecognizable character
: An invalid character was found in the input stream.

redefinition of #define symbol
: A symbol was previously defined, and cannot be re−defined until it has been undefined with **#undef**.

illegal digit in octal constant
: A character that was not between 0 and 7 inclusive was found in an octal constant.

more than one character in constant
: More than one character appeared between single quotes (e.g.: 'abc').

unterminated comment in file
: The end of the source file was reached before a comment ended.

unterminated #if or #ifdef in file
: The end of the source file was reached before the corresponding **#endif** was found.

Parser warnings

**** Internal Error ****
: An internal error occurred while parsing the source file. Please contact Introl.

non−prototyped parameter declaration
: A function was called before its prototype was seen.

illegal combination of pointer and integer
: Either a pointer was converted to an integer, or vice versa.

zero sized structure member
: A structure contains an element of size 0. This could be as a result of either an array of zero elements, or because an element used an undefined type.

illegal pointer combination
: Pointers to two different types were either compared or assigned.

storage class different in previous declaration
: A function or variable was declared twice with a different storage class each time.

implicit declaration of function
: A function was called before a prototype was seen.

expression with no effect, ignored

An expression with no effect was found and ignored.

semicolon or comma ignored

An extra comma or semicolon was found and ignored.

no return value for non−void function

A return statement was found without a return value in a function which was declared non−void.

constant value too large

A constant was found that was too large to fit in the type to which it was assigned.

symbol defined but not used in block

A symbol was defined but never used.

folded constant value too large

A constant expression may have been too large to fit in its desired type.

call to non−prototyped function

A call was made to a function which did not have a prototype.

zero or negative subscript

A zero or negative subscript was used in an array definition.

converting far pointer to near

A far pointer was converted to a near pointer.

Preprocessor errors

illegal #define

A syntax error was found in a **#define** directive.

cannot parse #define line

A syntax error was found in a **#define** directive.

illegal include

A syntax error was found in a **#include** directive.

illegal #line

A syntax error was found in a **#line** directive.

illegal #undef

A syntax error was found in a **#undef** directive.

unrecognizable preprocessor directive

An unrecognizable preprocessor directive was found.

cannot open #include file

The file specified with **#include** cannot be opened.

bad #if or #line expression

A syntax error was found in the **#if** or **#line** expression.

bad #if syntax

A syntax error was found in the **#if** directive.

string too long, truncated at right

A string constant was found that was too long and was truncated.

no matching #if for #endif

An **#endif** was found with no corresponding **#if**.

illegal #else

An **#else** was found with no corresponding **#if**.

too many nested #ifs

Too many levels of **#if**s were found (limit = 10).

missing """

Expression was missing a """

missing """ or character constant too long

Either a """ was missing, or a character constant was too long.

unexpected end of file, unbalanced #if, #ifdef or #ifndef

The end of the source file was found before all **#if** type expressions had been closed with their corresponding **#endif**.

unmatched paren or quote in macro call ... end of file
The end of the source file was found before a matching parenthesis or quote was found in a macro call.

too many #define parameters
More than 32 parameters were found in a macro definition.

improperly formed floating exponent
A floating−point constant had a malformed exponent.

incorrect number of parameters to macro
A macro was called with an incorrect number of parameters.

too many U's or L's in integer constant
Too many **U**'s or **L**'s were found in an integer constant.

recursive macro expansion
A macro resulted in recursion. This can only happen when the **−s** option is used.

can't open T file for writing
An intermediate temporary file could not be opened for writing.

can't open SYM file for writing
An intermediate temporary file could not be opened for writing.

Parser errors

arithmetic type required
A non−integer type was found where an integer type was required.

bad declaration
A syntax error was found in a declaration.

extern or static identifier needed
An **extern** or **static** modifier was needed but not found on a symbol.

cannot initialize
An illegal initialization was found.

function required
An attempt was made to call a non−function.

bad initializer expression
A syntax error was found in an initialization expression.

bad &
An '**&**' address−of operator was found in a bad context.

out of memory
The compiler ran out of memory.

cannot take address of register variable
The address of a variable declared with the **register** modifier cannot have its address determined.

bad break
A **break** statement was found in an invalid context, not in a **while**, **do**, or **for** loop or a **switch** statement.

case not in switch
A **case** statement was found outside a **switch** statement.

bad cast
A typecast was found in a bad context, or a syntax error was found in the typecast.

duplicate case expression
Each **case** statement in a **switch** statement must have a unique expression.

bad continue
A **continue** statement was found in a bad context.

default not in switch
: A **default** case was found outside a **switch** statement.

too many default statements
: More than one **default** statement was found for a **switch** statement, and only one is allowed.

not member of struct/union
: A symbol which does not exist in the given **struct** or **union** was referenced.

illegal operand type
: An expression contained values of non−compatible types.

return type cannot be function or array
: Arrays or functions are not valid return types.

illegal switch type
: **switch** statements must be on an integer type.

illegal structure reference
: A symbol was treated as a structure when it is not defined as one.

pointer on right of assignment op
: Two pointers cannot be added together.

struct (union) tag now specified as union (struct)
: A structure tag was later defined as a union or vice−versa.

integer type required
: An integer type was required in this expression.

lvalue required
: This expression was missing a left side.

const used as lvalue
: A constant was used as the left side of an expression.

right side of assignment op must be integer
: The value to be assigned needs to be an integer in this context.

missing member name
: The member name was missing from a structure expression.

no structure definition
: **struct** was found with no structure definition.

declared struct (union), defined union (struct)
: A structure was later defined as a union or vice−versa.

no enumeration definition
: **enum** was found with no enumeration definition.

pointer type required
: A pointer type was required in this expression.

sizeof returns zero
: This type is zero sized.

can't take size of object
: The size of this object cannot be determined (it was probably declared as extern).

unexpected identifier
: An identifier was encountered in an abstract declarator.

type found in old−style parameter list
: A parameter list in an old−style (K&R) function declaration contains a type specifier.

illegal array reference
: A symbol was referenced as an array and probably isn't one.

unexpected end of file
: The end of the source file was found before the end of an initializer was reached.

type mismatch from previous declaration
: A symbol was previously defined as another type.

multiple symbol definition

A global or static symbol was defined more than once.

missing parameter name
A parameter name was missing in a function prototype.

integer constant expression required
An integer constant expression was required in this context.

illegal operand types for operator
This expression has the wrong operand types for the result type.

declaration of parameter not in parameter list
A parameter was declared in the old−style function declaration that did not appear in the parameter list.

error writing file
An error occurred while writing a temporary file.

undefined operator on pointer type
An illegal operator was used on a pointer type.

symbol used but not defined in block
A symbol was referenced but never defined in this block.

undeclared identifier
A symbol was referenced but never defined in this block.

'&' added to identifier
A symbol was initialized with another simple symbol expression.

missing parameter type
The type of a parameter was left out of a function prototype.

operation on object of unknown size
An expression used an object of unknown size; probably because of an undefined type.

illegal type combination
An illegal combination of types was used in an expression.

expected token not found, inserted
A token was expected but not found, so it was temporarily inserted so parsing could continue.

definition of object with unknown size
An object was defined with an unknown type.

illegal type conversion
An object was converted illegally to another type.

too few parameters to function
A call was made to a function with too few parameters.

too many parameters to function
A call was made to a function with too many parameters.

parameter declaration different from prototype
A function prototype was seen that differs from the function declaration.

identifier used in a bad context
An identifier was seen in an invalid context.

zero sized field
A zero sized bit field was found.

field width too large for type
A bit field was too large for its underlying type.

extra comma in parameter list
A comma immediately follows a comma in a function parameter list.

unexpected token found
A token was found in a context that it should not be.

bad underlying type for field
A bit field was declared with an illegal underlying type.

bad type for struct/union member

A **struct**/**union** member was declared with an invalid type.

only first dimension may be empty

An array declaration had a empty dimension for a field other than the first.

duplicate member name

A **struct**/**union** was declared with more than one member with the same name.

divide by zero

A divide by zero was found in an expression.

mod by zero

A mod by zero was found in an expression.

string too long for initializer

A string that was too large for initialization was found.

too many initializers

Too many initializers were found for this expression.

The Linker

The Introl linker, **ild**, is used to combine multiple object files along with any library functions the object files might reference. The linker also acts as a *locator* to position program code and data in the target system's memory space. This chapter explains the linking process: how to use the linker and how to create linker command files.

Linker command line
: How to run the linker.

The linking process
: What the linker does and how it does it.

Linker command files
: How to make a linker command file to control the linking process.

Linker commands
: All the linker commands.

Partial linking
: How to use the linker to build a program in several passes.

Linker listing formats
: What the linker listing files looks like.

Warnings and errors
: Warning and error messages produced by the linker: what they mean and how to correct the errors.

Linker command line

```
ild −gXX [{options}] {filename}
ild05 [{options}] {filename}
ild08 [{options}] {filename}
ild09 [{options}] {filename}
ild11 [{options}] {filename}
ild12 [{options}] {filename}
ild16 [{options}] {filename}
ild68 [{options}] {filename}
```

The linker can be invoked with any of the names given above. **ild** with a **−g*XX***, where *XX* is a two digit processor number, is identical to using one of the **ild*XX*** forms. You may need to specify a **−g*XX*** option on the linker command line if you want to load a library for a processor that is not the default. A library added to a link session with a **−l** option (either on the linker command line or in a linker command file) will append the processor digits to the library name:

```
ild11 −o test test.o11 −lmylib
```

will link *test.o11* against the library *libmylib.a11*, while

```
ild11 −g03 −o test test.o11 −lmylib
```

will link *test.o11* against the library *libmylib.a03*.

Introl−CODE comes with libraries for various processors. The following table lists the **−g*XX*** options allowed and the processor supported:

Option	Processor
−g00	68000
−g01	6801
−g03	6301
−g09	6809
−g10	68010
−g11	68HC11
−g12	68HC12
−g16	68HC16
−g20	68020
−g40	68040
−g68	68332, etc.

Note that is possible, although generally undesirable, to link object files for different processors together. The linker does not distinguish between processor types when linking.

−a

Create an *absolute* file (default). This will also cause the linker to print errors about undefined

references. Relocation information is removed from the object file. This option is mutually exclusive with the **−r** option. Absolute files are executable. Some CODE programs like the debugger and ihex (a hex file conversion utility) require their input files to be absolute.

−A name[=value]
Define a macro for the linker command file preprocessor. If *value* is not given, *name* is defined as "1".

−b filename
Read in the file named filename. If the linker cannot open *filename* as specified, it will search for it in *$INTROL/lib*.

−L pathname
−d pathname
Add a path to the list searched when looking for libraries specified with the **−l** option, for linker command files specified with the **−g** option, or machine code description files specified with the **−M** option.

−B
Add information about program basic blocks to the listing output. This option works only when doing **icode** code generation.

−C
Add the program function call tree to the listing output. This option works only when doing **icode** code generation.

−D
This option disables any **dorigin** subcommands in the linker command file. This option is available for backward compatability with programs that do not understand the device origin.

−E
The linker will compile the machine code description file specified with a **−M** option and create a file called ***processor*.ccd** containing the result.

−F
Add program function information to the listing output. This option works only when doing **icode** code generation.

−f name
Set the name of the listing file to *name*. **−m**, **−u**, and **−y** are enabled automatically if none of the listing options (**−m**, **−u**, **−y**, **−B**, **−C**, **−F**) are given explicitly.

−g[processor][string]
The **−g** option can do two things: if a two−digit processor identifier is specified, *processor* is appended to the names of the library names specified with the **−l** option. If *string* is specified, then ***string*.ld** is specified as a linker command file, and is opened at that time. *processor* and *string* may be specified at the same time.

−k
Display the linker's name and version.

−lname
Expands to *lib**name**.a* where *name* can be any string. The file is searched as soon as the option is found, so placement on the command line is significant: a library should follow object files that reference any of its members. If the **−g** option was specified with a processor number, or if **ild** was invoked with the name **ild*XX***, then **−l*name*** is expanded to **lib*name*.a*XX***.

−m
Include a section map in the listing file.

−M processor
processor specifies a machine code description file to be used for icode link time code generation. The linker will try to open the compiled description file ***processor*.ccd**. If that fails the linker will open the description file ***processor*.mcd**. This option applies during **icode** code generation only.

−N list

Disable certain **icode** code generation optimizations. **list** is a list of characters that disable optimizations:
C − disable common subexpression elimination.; **S** − disable placing local variables of functions at static nesting levels in fixed RAM; **A** − disable all optimizations.

−o filename
Create an output object file having the name *filename* rather than the default name of **a.out**.

−O
Use the old, non−preprocessor, command file reader. This option is for backward compatability with pre 4.00.2 versions. This option should only be needed if you run into a problem using the new reader.

−p symbol
Add *symbol* to the undefined symbol list. See the import command.

−P
Read the default linker command file, normally specified by a **−gn*XX*** option, from standard input instead of **n*XX*.ld**.

−r
Retain *relocation records*. Relocation records must be kept if the output of the linker is to be reused as input to a subsequent run of the linker, see Partial linking. In this mode, the linker does not complain about unresolved references. This option is mutually exclusive with the **−a** option.

−s
Generate an ICOFF file without a symbol table. This option is ignored if the **−r** option is used.

−S
Add **icode** code generation statistics to the listing output. This option works only when doing **icode** code generation.

−t
Do not complain about **comm**symbols of the same name having different sizes.

−u
Include a list of files and their types in the listing file. The type can be: linker command file, object file, or library. If the type is library, the names of the modules extracted from that library will be included.

−w
Do not warn about relocations that overflow. The linker understands 8−, 16−, 20−, and 32−bit relocation types. This option causes the linker not to warn about relocation expressions that exceed the target size.

−x
Create an object file even if unresolved symbols are present.

−y
Include a symbol table in the listing file.

The linking process

Linking is the process of combining relocatable ICOFF object files together and producing a single file containing an executable image of the program that will ultimately be run on your target system. The object files read by the linker may have been produced either by the assembler or by a previous run of the linker, see Partial linking.

Command files, object files and libraries

The linker reads three types of files: ordinary text files, relocatable object files, and libraries. The linker determines a file's type by looking for specific information in the beginning of the file, rather than by its name. These files are processed by the linker in the order they are specified on the command line or specified

in linker command files.

Linker command files

If an input file is an ordinary text file, the linker processes its contents as a command file. You should specify your linker command file before any object files because the linker reads files in the order they are specified on the command line.

Relocatable object files

The linker adds the code and data from relocatable object files to the output file according to the rules contained in the linker command file(s). Therefore, all the definitions made in linker command files must usually be read by the linker before it sees any relocatable object files.

Libraries

A *library* is a collection of relocatable object files in one large file called an archive. The linker scans a table of contents in each library for unresolved symbols, extracting modules as required, until no more symbols can be resolved from that library. Since this scanning occurs only once for each specified library, the libraries should be specified last on the linker command line.

Linker command files

Linker command files are text files that contain, among other things, commands that control the linking process. Without a linker command file, the linker has a set of default rules that take effect. These default rules are generally of limited use and, in practice, at least one linker command file is usually needed to use the linker. The linker command files supplied with and generated by CODE are an important reference to help you configure the runtime environment.

A linker command file can contain three different elements: command line options, commands and filenames. In addition to defining where a program's code and data should reside in the target system's memory, allowing command line options and filenames in a linker command file can help get around line length limitations on systems such as MS−DOS.

A linker command file may also have comments. Any text delimited by **/*** and ***/** is ignored and any text following **//** is ignored until the end of the line that the **//** occurs on.

Linker command files are preprocessed using a C−like preprocessor. You can use **#define**, **#if**, etc. in a linker command file just as you can in a C file.

The complete syntax for linker command files is described below and in the following sections describing the linker commands. The syntax is designed to be very loose with many elements, especially semicolons, being optional. We recommend, however, that you always use semicolons to avoid possible problems.

Some linker commands require that expressions must have known values at the time of their use (either from an object file, or from the set or let commands). Symbols that have not as yet been defined (they may be in an object file that has not been read yet) are not allowed in these expressions.

Options

Command line options can appear in linker command files. They have the same effect they have on the command line. For example, it is common for the **−o** option to appear in a linker command line as in:

```
ild12 −o test −gn12 test.o12 −lc
```

This line in a linker command file would cause the output file produced by the linker to be named *test*:

```
−o 'test'
```

The single quotes around *test* insure that the filename is not interpreted as a linker command. See the section on filenames for more information.

Expressions

Some linker commands take expressions as aguments. The following operators, listed in order of highest to lowest precedence, can be used in these expressions:

endof(name)
: Returns the address of one byte past the end address of the output section *name*. It is only meaningful after that section has been assigned an origin.

sizeof(name)
: Returns the size in bytes of the output section *name*.

startof(name)
: Returns the starting address of the output section *name*. It is only meaningful after the given section has been assigned an origin.

− ! ~
: Arithmetic negate, logical negate, and one's complement. The result of the unary − operator is the negative of its operand. The result of the logical negation operator (**!**) is 1 if the value of its operand is 0, and 0 if the value of its operand is non−zero. The **~** operator yields the one's complement of its operand.

*** / %**
: Arithmetic multiply, divide, and mod. The binary ***** operator indicates multiplication. The binary **/** operator indicates division. The binary **%** operator indicates the remainder from the division of the first operand by the second.

+ −
: Arithmetic add and subtract. The binary **+** operator yields the sum of its two operands. The binary **−** operator yields the difference of its two operands.

>> <<
: Shift right and shift left. The result of *expr1* **<<***expr2* is *expr1* left shifted *expr2* bits, with the vacated bits getting zero filled. The result of *expr1* **>>** *expr2* is *expr1* right shifted *expr2* bits, with the vacated bits getting filled by the sign bit.

< > <= >=
: Logical less than, greater than, less than or equal to, and greater than or equal to. These operators all yield 0 if the condition is false, and 1 if it is true.

== *!*=
: Logical equals and logical not equals. These operators all yield 0 if the condition is false, and 1 if it is true.

&
: Bitwise and. The result is the bitwise AND function of the operands.

^
: Bitwise exclusive or. The result is the bitwise exclusive OR function of the operands.

|
: Bitwise inclusive or. The result is the bitwise inclusive OR of the operands.

&&
: Logical and. Returns 1 if both operands are non−zero, and 0 otherwise.

||
: Logical or. Returns 1 if either of the operands are non−zero, 0 otherwise.

Filenames

Any word in the outer context of a linker command file that is not recognized as a valid command will be treated as a filename that the linker should read. As mentioned earlier, the file can either be an ordinary text file treated as another linker command file, a relocatable object file or a library file. You can specify a name of a file which conflicts with a linker command or which contains special characters (such as ``+" or ``/" which are part of the linker command file's expression syntax) by enclosing the name in single quotes. For example, the following linker command file specifies three filenames:

```
file1.o
'file2.o'
'path/file3.o'
```

In this case, the second filename did not need quoting. Names containing backslash characters ('\') also need to be quoted.

Linker Commands

Commands in a linker command file direct the linker to perform certain actions, such as defining memory regions, defining symbols, and doing link time constraint checking. Commands may be in either uppercase, lowercase, or a combination. The following section describes the commands which may be used in a linker command file.

align
: Specify section and group alignment.

bank
: Specify a bank number for paged memory.

bss
: Mark a group or section as blank static storage.

check
: Conditionally generate an error.

comm
: Mark a section as a common section.

comms
: Specify that a section should contain comm variables.

copiedfrom
: Specify another section from which the present section should be filled.

data
: Mark a section as a data section.

dorigin
: Specify the device origin of a section.

duplicateof
: Duplicate the contents of another section in the current section.

eeprom
: Mark as window or group as residing in EEPROM.

end
: The end of the linker command file.

expand
: Expand a section withing a group.

export
: Export selected symbols in a linked file.

fatal
: Mark a check error as fatal.

filealign
: Specify the alignment of sections in the output file.

fill
: Fill a section or group with a value.

group
: Define a group of sections.

import
: Force the importing of a symbol from a library.

io
: Mark a window or group as containing I/O device registers.

itemalign
: Specify the alignment of input sections in output sections.

let
: Define a symbol.

long
: Define a long value in the optional header.

magic
: Change the output file's magic number.

maxbuffers
: Control linker memory usage.

maxsize
: Specify the maximum size of a group or section.

minsize
: Specify the minimum size of a section.

mode
: Specify the file mode bits for the output file.

nofilhdr
: Generate an output file without a file header.

nosections
: Generate an output file with no section headers.

opthdr
: Generate an optional header in the output file.

origin
: Specify the origin of a group or section.

private
: Specify that all symbols in the output file are not to be exported.

raise
: Move a section to the top of a group.

ram
: Mark a window or group as residing in read/write memory.

readline
: Read and process the files on the linker command line.

rom
: Mark a window or group as residing in read−only memory.

section
: Define an output section.

set
: Define the value of a linker variable.

short
: Define a short value in the optional header.

string
: Define a string value in the optional header.

text
: Mark a section as an executable code section.

window
: Associate a group with a memory bank.

align

section *name* align *expr*

The **align** subcommand causes the linker to align the beginning of a section on a memory boundary that is divisible by the value of the expression *expr*. In general, it only makes sense to use the **align** command on sections that do not contain an explicit origin subcommand (since an explicit origin implies a particular alignment). For example, given the following linker command file:

```
section .abc origin 0x1000 = .abc;
section .def align 2 = .def;
```

the linker will start the section named *.def* on an even boundary. Note that if an alignment is required, which would happen in this case if the size of *.abc* were odd, the hole byte between these two sections would not be represented in the resulting object file at all. The linker performs an alignment adjustment by adding an appropriate offset to the section's origin address rather padding the previous section.

bank

group *name* window *windowname* [bank *expr*]

The **window** subcommand associates a group with a window for bank switching. The window must be defined before a **group** command refers to it. The optional **bank** subcommand tells the linker to use *expr* as the bank identifier rather than the default internally generated sequential number.This could be used, for example, to make the bank identifiers correspond more closely with the underlying hardware bank switching mechanism. For example the hardware might allow eight banks, each selected by a bit in an i/o port. The bank identifiers for a window like that might be 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, and 0x80. Bank identifiers can be any value between 0 and 4096 (12 bits).

Bank identifiers for all groups in a window must be unique.

bss

A **bss** (blank static storage) group or section is one in which there is no downloadable data.

group *name* bss

The **bss** subcommand causes the sections defined in the group to have the bss attribute set.

section *name* bss

The **bss** subcommand causes the linker to produce a BSS (empty) section. This subcommand is not normally needed for sections that receive any input sections since the linker will automatically create an output section as a BSS section if all of its input sections are BSS sections. We recommend that you use the **bss** subcommand anyway for all sections which you want to be generated as BSS sections since it will allow the linker to report an error if any of the input sections are not BSS sections.

Padding sections that do not take data from any input sections need the **bss** subcommand, otherwise the linker will fill the section with zeroes.

check

check *expr* [fatal] *ident*

The **check** command is used to produce user defined error or warning messages. If the expression, *expr*, generates a nonzero value, the linker will generate a warning message, displaying the text of *ident*. If the word **fatal** follows the command's expression, the linker generates a fatal error message rather than a warning. The *ident* almost always should be a string of text enclosed in single quotes.

This command is typically used to check that specific memory regions have not been overfilled. For example, if you wanted the linker to generate an error message if the section named *.abc* contained more than 32K bytes of data, you could use the following check command:

```
check sizeof(.abc) > 32767 fatal "section .abc is too large (> 32K)";
```

Since the linker does not evaluate the expressions of **check** commands until the linking process is complete, their order in the linker command file does not matter.

comm

section *name* comm

The comm subcommand generates a *common section*. The linker normally creates an output section by concatenating the data from many input sections. A common section, however, is only as large as its biggest input section since the linker origins every input section to offset zero, that is, the sections overlay each other.

comms

section *name* comms

The **comms** subcommand tells the linker that the output section containing it is eligible to receive data defined with the assembler's **comm** directive. Normally, you can just place a **comms** subcommand on one of your RAM sections. If none of your object files contain any data generated by the assembler's **comm** directive, you do not need to put the **comms** subcommand on any of your sections. If the linker encounters an object file that contains **comm** generated data and you do not have any sections that contain the **comms** subcommand, the linker will generate an error.

copiedfrom

section *name1* copiedfrom *name2*

copiedfrom allows you to create a section whose data comes from another output section. Given the linker command file:

```
section .abc origin 0x1000 = .xyz;
section .def origin 0x8000 copiedfrom .abc;
```

The linker will copy the data from output section *.def* into *.abc*, and will transform section *.def* into a BSS section. This subcommand is called **copiedfrom** because the data copied by the linker from *name1* to *name2* can be copied back from *name2* to name1 at run time.

For an example of handling initialized data in C, consider the following linker command file:

```
/* ROM at 0x1000 */
section .text origin 0x1000 = .text;
section .const = .const;
section .strings = .strings;
section .Rdata;
/* RAM at 0x8000 */
section .data origin 0x8000 copiedfrom .Rdata = .data;
```

The data in section *.data* is copied into the section *.Rdata* at link time and is placed into ROM. At run time, suitable startup code copies the data from *.Rdata* into the BSS section *.data* at address 0x8000 in RAM for execution. Our sample startup code and linker command file uses this method to handle initialized data in C.

The **copiedfrom** subcommand is also very useful when building a *paged memory system* (i.e. bank switching). Given a paged system with the following characteristics:

- 16−bit address space;
- Global ROM from 0x4000 to 0x7fff;
- 8K paged memory region from 0x8000 to 0x9fff;
- A 256K by 8 EPROM with appropriate mapping hardware such that any 8K aligned region of 8K at or above offset 0x10000 within the EPROM can be mapped into the paged region at 0x8000;
- A set of input files containing executable code that is to run in the 8K paged region at 0x8000 but is to be placed in the EPROM; and
- The code that is to be placed in each region (bank) of the EPROM has been segregated into sections with names of the form *.bank0*, *.bank1*, *.bank2*, etc. (using the compiler's −a option, most likely).

The following linker command file fragment would handle the first 4 banks:

```
section .Rbank1 origin 0x10000;
section .Rbank2 origin 0x12000;
section .Rbank3 origin 0x14000;
section .Rbank4 origin 0x16000;
section .bank1 origin 0x8000 copiedfrom .Rbank1 = .bank1;
section .bank2 origin 0x8000 copiedfrom .Rbank2 = .bank2;
section .bank3 origin 0x8000 copiedfrom .Rbank3 = .bank3;
section .bank4 origin 0x8000 copiedfrom .Rbank4 = .bank4;
```

It should be obvious how to extend this technique to other similar systems; add more similar sections and set their origins appropriately. Notice that the data from all the banks have been linked at the common paged region address since that is where the code is actually executed. Furthermore the term **copiedfrom** makes sense from the standpoint of the running system since, for example, data for the output section *.bank3* is

effectively copied from the section *.Rbank3* by the mapping hardware (of course, unlike the example of initialized data in C, the memory mapping hardware does not actually copy any data).

Again, the naming convention used in the previous example is meant to imply that a section whose name contains the *R* resides in the EPROM.

data

section *name* data

The **data** subcommand cause an appropriate ICOFF section flag to be set for the section containing the subcommand. The **data** flag is used to indicate a section containing data (non−executable). The interpretation of this flag within an ICOFF file is dependent on the program that ultimately reads the file. Introl's debugger, for example, uses the fact that a section only contains data (i.e. was defined with the **data** subcommand) to set breakpoints more efficiently on certain targets.

If these flags are required by an Introl utility, the documentation for the utility in question will say so.

dorigin

group *name* dorigin *expr*

The **dorigin** subcommand sets the *device origin* for a group. The device origin is used to position a group in a physical device such as an EEPROM, and is typically used in bank switched systems. The following example has a fixed memory region from 0xC000 through 0xFFFF and a page window from 0x8000 through 0xBFFF. The **dorigin** subcommands cause the fixed memory region to be placed in the device at physical address 0x0000 and the paged areas to be placed in 16K areas above the fixed area. The **ihex −d** option allows you to create hex files using the device origin.

```
// This is the un−paged memory area from 0xC000 to 0xFFFF

group ROM rom origin 0xC000 maxsize 0x10000 − 0xC000 dorigin 0;

... root sections

group ROM;

// The paged memory area from 0x8000 to 0xBFFF

window PPROG rom origin 0x8000 maxsize 0xC000 − 0x8000;

group BANK0 window PPROG dorigin 0x04000;
section .bank0 = .ftext;
group BANK0;

group BANK1 window PPROG dorigin 0x08000;
section .bank1 = .ftext;
group BANK1;

group BANK2 window PPROG dorigin 0x0C000;
section .bank2 = .ftext;
group BANK2;

group BANK3 window PPROG dorigin 0x10000;
section .bank3 = .ftext;
group BANK3;

// Other banks can be added as needed
```

duplicateof

section *name1* duplicateof *name2*

duplicateof allows you to create a section whose data is duplicated from another output section. Given the linker command file:

```
section .abc origin 0x1000 = .xyz;
section .def origin 0x8000 duplicateof .abc;
```

The linker will duplicate the section information from section *.abc* into *.def*.

eeprom

The **eeprom** subcommand can be used to set the *device type* of a window or group to be EEPROM. The device type is for informational purposes only.

window *name* eeprom

The **window** command is used to associate groups into a defined memory window. The **eeprom** subcommand is optional and is implicitly added to each group associated with a window.

group *name* eeprom

This subcommand defines the device type of the group as EEPROM.

end

This command indicates the end of the linker command file. Any text following the **end** command is ignored.

expand

section *name* expand

The **expand** subcommand, allowed only on sections within a group, tells the linker that this section should be expanded so the the sections in a group will fill the entire group. An example of **expand**:

```
group RAM bss origin 0x2000 maxsize 0x8000;
section .bss = .bss;
section .heap expand minsize 2048;
section .stack minsize 1024;
group RAM;
```

This set of commands defines a 32K RAM area starting at 0x2000. The group is marked **bss**, meaning it is uninitialized. The (fixed size) section *.bss* is the first section in the group followed by *.heap* with the **expand** flag set followed by *.stack* with a minimum size of 1024 bytes.

The final size of the *RAM* group will be exactly 32K: *.heap* will be expanded, pushing *.stack* up to the end of *RAM*, so that *.heap* fills the entire unused RAM area.

export

export *symbol*{,*symbo*l}

This command makes the named symbols that are local to the final output file exported. The named symbols are changed from **IC_STAT** to **IC_EXT**. This can be used to control access to symbols in a linked file. See also **private** below.

fatal

check *expr* [fatal] *ident*

The **check** command is used to produce user defined error or warning messages. If the expression, *expr*, generates a nonzero value, the linker will generate a warning message, displaying the text of *ident*. If the word **fatal** follows the command's expression, the linker generates a fatal error message rather than a warning. The *ident* almost always should be a string of text enclosed in single quotes.

This command is typically used to check that specific memory regions have not been overfilled. For example, if you wanted the linker to generate an error message if the section named *.abc* contained more than 32K bytes of data, you could use the following check command:

```
check sizeof(.abc) > 32767 fatal "section .abc is too large (> 32K)";
```

Since the linker does not evaluate the expressions of **check** commands until the linking process is complete, their order in the linker command file does not matter.

filealign

filealign *expr*

Even though Introl's ICOFF object file format was not designed to be compatible with the COFF format, ICOFF files are compatible with COFF in a limited way. In particular, an ICOFF file without a symbol table or any normal relocation records (i.e. an absolute file produced by the linker) will be loadable by many standard COFF program loaders (such as many SystemV Unix kernels).

The **filealign** command is strictly related to COFF compatibility in that it causes the beginning of the actual data for each section in the ICOFF file to be aligned on a file offset boundary that is divisible by *expr*. This command is not normally used. The position of the **filealign** command in the command file does not matter since it only effects the linker's final generation of your object file.

fill

group *name* fill *expr*

The **fill** subcommand sets the default fill value for sections in a group.

section *name* fill *expr*

The **fill** subcommand changed the default value used to pad empty spaces in sections with a the value *expr*. All empty bytes in the section will be filled with *expr*.The default **fill** value is 0 (zero).

fill can be used in conjunction with **group** and **expand** to fill empty spaces with a known value:

```
group ROM origin 0xe000 maxsize 8192;
section .text text = .text;
...
section .fill expand fill 0xFF;    // fill empty space with 0xFF
section .vectors origin 0xffc0;
group ROM;
```

group

group *name* [{*subcommand* [*arg*]}]

The **group** command is used to place sections, defined with the **section** command, into related *groups*. There are several advantages of placing sections into groups:

- The physical placement of sections and their interrelationship can be documented in the linker command file.
- Groups of sections correspond more closely with the environment that exists in most embedded systems, groups can represent devices such as ROMs and RAMs.
- The linker can do more error checking on groups of sections because it knows the exact boundaries of the area into which the sections should fit.
- A section can be expanded to take up unused group space with **expand**.
- Groups can be associated with a window to facilitate *bank switching*.

The **group** command does not create anything explicitly in the file output file except as the result of the **fill** subcommand.

The **group** command places all following sections in the group *name* until another **group** command is encountered. The best way to mark the end of a group is with an empty **group** command with the same name:

```
group ROM origin 0xe000 maxsize 8192;
section .text text = .text;
...
group ROM;    // End of the ROM group
```

import

import *symbol* {, *symbol*}

Add the given symbols to the list of undefined symbols. An undefined symbol error will occur if the any of the symbols are unresolved during linking.

io

The **io** subcommand can be used to set the *device type* of a window or group to be Input/Output and peripherial registers. The device type is for informational purposes only.

window *name* io

The **window** command is used to associate groups into a defined memory window. The **io** subcommand is optional and is implicitly added to each group associated with a window.

group *name* io

This subcommand defines the *device type* of the group as I/O registers.

itemalign

Normally, when the linker has determined an output section into which an input section should be placed, it concatenates the data from the input section into the part of the output section that it has already built. If the output section contains an **itemalign** subcommand, the linker will insert padding bytes of zeroes in the output section so that each new input section that is added will begin on a memory boundary divisible by the value of the expression *expr*.

group *name* itemalign *expr*

The **itemalign** subcommand set the default **itemalign** value for the sections in a group.

section *name* itemalign *expr*

As a complete example of the operation of **itemalign**, consider the following three object files and their sections' contents:

file name	section name	size in bytes
test1.o	.abc	0x5
test2.o	.abc	0x10
test3.o	.abc	0x5

If these files are linked using a linker command file named *test.ld* containing:

```
section .abc origin 0x1000 itemalign 2 = .abc:
```

using the command line:

```
ild68 test.ld test1.o test2.o test3.o
```

the linker would produce a file with the following memory layout:

Address range	No. of bytes	Input section name	From file	Output section name
0x1016−0x101a	0x5	.abc	test3.o	.abc
0x1006−0x1015	0x10	.abc	test2.o	.abc
0x1005−0x1005	0x1	(none) Padding byte inserted by the linker		
0x1000−0x1004	0x5	.abc	test1.o	.abc

Since the input section *.abc* from file *test3.o* was the last input data from *.abc* read by the linker, it did not put a padding byte after that section at address 0x101b. There would be a padding byte at 0x101b if any more input files were read that contained data in the section *.abc*.

let

let *symbol* = *expr*

The **let** command is used to define new global symbols. It is virtually identical to using the equ directive in an assembler source file and exporting the equated symbol. For example, the line:

```
let label1 = 10;
```

in a linker command file would have the effect of defining a global symbol named *label1* and setting its value to 10 just as would the following assembler code fragment:

```
label1                  equ                     10
                        export                  label1
```

This assumes, of course, that the object file produced by assembling the previous code fragment is actually linked into your program.

The expressions used in the **let** command are evaluated before linking has actually started since the normal reason for defining such symbols is that they be referenced from one or more of the relocatable object files. Since these expressions are evaluated before linking begins, they can not include any components whose values are not known in advance; this includes origins of sections that do not contain an explicit **origin** subcommand and also end addresses of any section.

See the **set** command for a way to assign values to symbols after linking is finished.

long

opthdr long {[*order*]} = *expr*

The **opthdr** command allows you to produce a data structure in the output object file called the optional header. The **long** subcommand is used to place a 32 bit value in the optional header in the byte order specified by the optional *order*.

```
opthdr long[0,1,2,3] = 1000;
```

The expressions in the **opthdr** command may use symbols that will be defined later in the link process.

magic

magic *expr*

The magic command causes a magic number equal to the result of the expression *expr* to be written to the file header of the output object file. By default, the value 0601 (octal) is used to indicate an ICOFF file. Files produced with a different magic number will not be usable by standard Introl object file tools.

maxbuffers

maxbuffers *expr*

maxbuffers controls the amount of RAM used by the linker. By default, the linker will use as much RAM as possible to buffer various parts of the object files. The linker will normally spill various of those parts to scratch files as required. When the linker encounters the maxbuffers command, it will allocate at most *expr* number of 4K buffers. A value of 0 will disable buffering and will cause the linker to use scratch files for all parts of the object file. This option may be useful if you are running on an operating system with virtual memory and want to limit the working set size of a large link run.

maxsize

The **maxsize** subcommand set the maximum size for a group or section. If no maximum size is specified the group or section grow essentially without bound.

group *name* maxsize *expr*

The purpose of **maxsize** is to specify the maximum amount of memory a group of sections may use.

The **maxsize** subcommand is required if you want to use the **section** **expand** subcommand for one or more sections in a group.

section *name* maxsize *expr*

The purpose of **maxsize** is to fill discontiguous memory regions with data from the same set of input sections. You could use **maxsize** to prevent overflows, but you should use **check** instead, since errors could otherwise be masked.

For example, if your system has 2 EPROMs into which you want to put code, but there is a hole in the memory space between them, you could use the **maxsize** subcommand on a pair of **section** commands to fill both of them.

It is legal for an input section to appear in more than one **section** command if the maxsize modifier is used on all previous **section** commands containing the input section. For example, if you have a system that contains an 8K EPROM at 0x1000 and an 8K EPROM at 0x8000 and you wanted the sections *.text* and *.const* to placed in them, you could direct the linker to fill the first EPROM and then to start filling the second EPROM when the first one was filled by using the following **section** commands:

```
section .eprom1 origin 0x1000 maxsize 8192 = .text, .const;
section .eprom2 origin 0x8000 maxsize 8192 = .text, .const;
```

As a complete example of the operation of **maxsize**, consider the following object files and their sections' contents:

file name	section name	size in bytes
text1.o	.abc	0x10
	.def	0x20
	.ghi	0x30
test2.o	.abc	0x100
	.def	0x200
	.ghi	0x300
	.klm	0x400
text3.o	.abc	0x50
	.def	0x60

If you were to link the files using a linker command file named *test.ld* containing:

```
section .eprom1 maxsize 0x116 origin 0x1000 = .abc;
section .eprom2 origin 0x8000 = .abc, .def, .ghi, .klm;
```

using the linker command line:

```
ild68 −gtest test1.o test2.o test3.o
```

The linker would produce a file with the following memory layout (note that the intent of this linker command file is to spread the section *.abc* through two discontiguous sections, but to place the rest of the sections in the program all in the same section):

Address range	No. of bytes	Input section name	From file	Output section name
0x89a0−0x89ff	0x60	.def	test3.o	.eprom2
0x8950−0x899f	0x50	.abc	test3.o	.eprom2
0x8550−0x894f	0x400	.klm	test2.o	.eprom2
0x8250−0x854f	0x300	.ghi	test2.o	.eprom2
0x8050−0x824f	0x200	.def	test2.o	.eprom2
0x8020−0x804f	0x30	.ghi	test1.o	.eprom2
0x8000−0x801f	0x20	.def	test1.o	.eprom2
0x1010−0x110f	0x100	.abc	test2.o	.eprom1
0x1000−0x100f	0x10	.abc	text1.o	.eprom1

Notice how the data from section *.abc* in the file *test3.o* was put in the *.eprom2* section. It was put there because the *.eprom1* section already contained 0x110 bytes of data and the *.abc* section from *test3.o* contains 0x50 bytes which would cause *.eprom1* to overflow its maximum size.

A few other important points arise from this example:

- The order in which the files are read by the linker can change the memory layout;
- An input section from a single object file is the basic atomic unit of linkage and is never split into pieces; it must be put in a single output section;
- The last output section that receives data from an input section whose name also appears in another output section's input list does not need a **maxsize**; and
- The linker uses a *first−fit* algorithm to pack data into output sections. If the linker has passed up an output section because its current input section would cause it to overfill, that same output section will be considered again. In other words, the linker will continue to try packing additional data in the stub ends of sections into which previously seen input sections did not fit.

minsize

section *name* minsize *expr*

The **minsize** subcommand effectively imposes a minimum size on a section. In practice, it is normally used for creating empty padding sections (such as for stacks, heaps, etc.). After all the object files and libraries have been read, any section whose size is less than its minimum size as specified by a **minsize** subcommand will be padded on its end with zeroes as necessary to meet the minimum size specified. If no **minsize** is used, no such padding will occur. For example, the **section** command:

```
section .abc minsize 0x1000 origin 0x100 = .abc;
```

will create a section named *.abc* located at address 0x100 and taking its data from input sections of the same name. If all the input object files and libraries have less than 4K of data in section *.abc*, the linker will pad the section enough zeroes to make the size equal to 4K. If there was more than 4K of data placed in the section, the **minsize** command will have no effect.

A padding section is typically made with a section command like:

```
section .stack bss minsize 0x2000;
```

The **bss** subcommand causes the output section to be a BSS section. Since the section has been specified as having the BSS attribute, the linker does not actually place physical bytes of 0 in the output section; it just adjusts the section's size as necessary.

There is one possible problem that can occur when using this **section** command in the previous example: if an input object file accidentally contains a section named *.stack*, the data from that section will be put in the output section. To protect yourself from such accidents, we recommend using a null input section list whenever you make an output section for which there are not supposed to be any corresponding sections. The previous **section** command would be ideally written as:

```
section .stack bss minsize 0x2000 = ;
```

This guarantees that absolutely no input sections will find their way into this output section (which, in this case, is what we want).

mode

mode *expr*

The mode command is used to set the file *mode* or permissions of the output file created by the linker.

nofilhdr

The **nofilhdr** command, as well as the following **nosections** command, cause a non−ICOFF file to be produced by the linker. The **nofilhdr** command causes the linker not to generate the ICOFF file header structure.

nosections

The **nosections** command causes the linker not to generate the section header data structures in the output file.

The **nofilhdr** and **nosections** commands are most often used together which, in effect, causes a file to be produced that just contains the raw binary data from all of the sections in the file. For example, given the following object file descriptions:

filename	section	size in bytes
file1.o	.abc	10
	.def	20
	.ghi	30
file2.o	.abc	100
	.ghi	200

and a linker command file named *lcf.ld* containing the following section commands:

```
nosections;
nofilhdr;
section .abc origin 0x1000 = .abc;
section .def origin 0x2000 = .def;
section .ghi origin 0x3000 = .ghi;
```

and linked with the command:

```
ild68 lcf.ld file1.o file2.o −o file3.o
```

the linker would produce a 360 byte file (named file3.o) whose first 110 bytes contained the data from the section .abc and whose next 20 bytes would contain the data from section .def and whose final 230 bytes would contain the data from section .ghi. These commands are most useful for files that contain a single section and allow you to easily get at the raw binary data for that section (with, for example, another program).

opthdr

opthdr *type* {[*order*]} = *expr* {, *type* {[*order*]} = *expr*}

The **opthdr** command allows you to produce a data structure in the output object file called the optional header. The optional header is a place where you can put extraneous information that will not be looked at by any Introl software (except for the −o option to idump, whose purpose is to examine the optional header). When used with **nosections** and **nofilhdr** commands, the optional header is at the beginning of the output file, allowing arbitrary binary data to be placed at the beginning of an object file.

Valid types are **short**, **long**, and **string**.

The syntax for the **opthdr** command is similar to that which is used in C to define a structure. Following the **opthdr** keyword you specify the data to place in the optional header. The command allows 16−bit integers, 32−bit integers and string data to be specified. Furthermore, the optional *order* can be used to specify the byte order use for 16 and 32−bit words. For example, to specify an optional header containing the following:

- The ASCII values of the characters in the word "test";
- A 16−bit integer whose value is 100, written in big endian byte order; and
- A 32−bit integer whose value is 1000, written in little endian byte order;

you could use the following **opthdr** command:

```
opthdr
        string = "test",
        short[1,0] = 100,
        long[0,1,2,3] = 1000;
```

The expressions in the **opthdr** command may use symbols that will be defined later in the link process.

origin

The **origin** subcommand tells the linker where a group or section should be placed in memory.

group *name* origin *expr*

If a group command has no origin then it starts immediately after the last group or section encountered in the linker command file. For example, consider the following **group** command:

```
group RAM origin 0x1000;
```

This group starts at address 0x1000.

section *name* origin *expr*

Without an **origin** subcommand a section will follow the previously defined section corresponding to the last **section** command the linker has seen. If there have not yet been any sections defined, the origin is 0 in the absence of an **origin** subcommand; but the linker will issue a warning in this case since it usually indicates a problem with your linker command file. For example, consider the following **section** command:

```
section .abc origin 0x1000 = .abc;
```

This section starts at address 0x1000. If the following two **section** commands were used:

```
section .abc origin 0x1000 = .abc;
section .def = .def;
```

and the section named *.abc* had a final size of 0x100 bytes, the section named *.def* would be located at address 0x1100.

The **origin** subcommand is not allowed in a group if the group has no **origin**. Further, the origin specified must be within the boundries specified by the enclosing group's **origin** and **maxsize**.

private

This command makes all exported symbols in the output file into file local symbols. Along with the **export** command, this allows you to control the symbols that a linked file defines.

raise

section *name* raise

The **raise** subcommand, allowed only on sections within a group, tells the linker that this section, and any sections above it, should be moved up in memory towards the top of the group. An example of **raise**:

```
group ROM origin 0xE000 maxsize 0x2000;
section .text = .text;
section .vectors raise;
group ROM;
```

This set of commands defines a 8K ROM area starting at 0xE000. The (fixed size) section *.text* is the first section in the group followed by *.vectors* with the **raise** flag set.

The final size of the *ROM* group will be exactly 8K: *.text* will be at the bottom of the ROM group and *.vectors* will be origined just below the top of the ROM group. The space between *.text* and *.vectors* will be empty

ram

The **ram** subcommand can be used to set the *device type* of a window or group to be read/write memory. The device type is for informational purposes only.

window *name* eeprom

The **window** command is used to associate groups into a defined memory window. The **ram** subcommand is optional and is implicitly added to each group associated with a window.

group *name* ram

This subcommand defines the *device type* of the group as read/write memory.

readline

Normally the linker processes the arguments from its command line in order from left to right. When a text file is encountered as a command line argument, it is completely processed from beginning to end as a linker command file. The **readline** command is used to cause a temporary diversion in this normal order of linker command processing. When the linker encounters the **readline** command in a linker command file, it saves its place in the linker command file that contained the **readline** command itself and begins processing that part of the command line which has not yet been processed. After processing the rest of the command line, the linker resumes its processing at the remembered position in the linker command file that contained the **readline** command. Typically, only one **readline** command will appear in any linker command file in a single run of the linker.

Using the **readline** command, you can create command files that allow the linker to be invoked similarly to traditional host system linkers such as the Unix **ld** linker. Our sample linker command file supplied with the compiler uses **readline**. Generally, one or more object files will be specified in a linker command file prior to the readline command. These object file(s) are typically the start−up code for your program as with our sample linker command file. The libraries are specified in the linker command file following the **readline** command. Using this design, the linker first reads the start−up code and then reads the rest of the command which contains your object files' names and finally re−enters the linker command file to resolve symbols from the libraries.

In summary, **readline** allows you to pinch the processing of the rest of the command line between a set of initialization code and your libraries.

For example, given a program made from two source files, *test1.c* and *test2.c*, you would first compile them, producing the object files *test1.o11* and *test2.o11*. The command line to link them, using our sample linker command file, would be:

```
ild11 −gc11 test1.o11 test2.o11 −o test
```

The first argument specified to the linker, **−gc11**, loads a sample linker command file, **c11.ld**. The linker will process it up to the **readline** command. The object files *test1.o11* and *test2.o11* will be processed following the **readline** command. Finally, the linker will return to processing the **c11.ld** file and pick up anything necessary from the libraries to link the program. Note that the −o command line option, in this case, is used to set the file name of the linked program to *test*.

rom

The **rom** subcommand can be used to set the *device type* of a window or group to be read−only memory. The device type is for informational purposes only.

window *name* eeprom

The **window** command is used to associate groups into a defined memory window. The **rom** subcommand is optional and is implicitly added to each group associated with a window.

group *name* rom

This subcommand defines the *device type* of the group as read−only memory.

section

section *name* [{*subcommand* [*arg*]}] [= [{*section* {,} }]]

The **section** commands are the centerpiece of any linker command file, since they describe the sections in your program and their layout in memory as well as their address in the actual object file which may be different than their address in memory.

Sections can be grouped into logically related groups, see the **group** command.

The purpose of a **section** command is to tell the linker to create a section in the output object file. There is a one−to−one correspondence between the sections in the file produced by the linker and to the **section** commands contained in the linker command file(s).

There are two major parts of a **section** command − the *output section* and zero or more *input sections*. The *name* argument is the output section. This is the section that is produced by the linker as a result of the **section** command. A minimal **section** command could, in fact, consist simply of the word section and the name of an output section, as described below; in practice, you will never use such minimal **section** commands. The input sections are specified by a comma−separated list of input section names following an =. The overall syntax of a **section** command's input sections and its output section is meant to be reminiscent of an assignment expression in C. For instance, in C, you would use the following expression:

```
a = b;
```

to assign the value of *b* to *a*; data is effectively copied from *b* to *a*. Similarly, a section command of the form:

```
section .abc = .def, .ghi, .jkl;
```

means that data from the sections named *.def*, *.ghi* and *.jkl* (the input sections) in the input files will all be placed in the section named *.abc* (the output section) in the linker's output file. The order in which the data from multiple input sections will be placed in an output section is dependent on the order in which the linker reads its input files. If you want all the data from a section to be contiguous, you must put it all in its own output section. For example, the following three **section** commands:

```
section .abc1 = .def;
section .abc2 = .ghi;
section .abc3 = .jkl;
```

are not equivalent to the previous single **section** command. This group of **section** commands causes three sections to be produced in the output file; each will contain data from a single input section. It is common to use the same set of names for both output and input sections; the previous three **section** commands would probably be written as:

```
section .def = .def;
section .ghi = .ghi;
section .jkl = .jkl;
```

which has exactly the same effect as the previous set except that the output sections' names reflect the name of the corresponding input section. Finally, you can leave out the input section list completely in this very common case. In the absence of an input section list, the linker will behave as if a **section** command contained an input section list containing a single section with the same name as the output section. For example, the previous three section commands could also be written as:

```
section .def;
section .ghi;
section .jkl;
```

In general the linker assigns input sections to output sections by first scanning the **section** commands it has seen so far. If there is a list of input sections to the right of an =, the linker will place those sections into the specified output section. If there is no =, the linker will assume that an input section is to be placed into an output section of the same name. If there is an = with no input section list, no input sections will be placed into that output section. Normally it is illegal for the same input section name to appear in more than one section command. For example, given the **section** commands:

```
section .abc = .def, .ghi;
section .pdq = .ghi, .xyz;
```

the second command is illegal since it specifies *.ghi* as an input section for section *.pdq* when it had already appeared in the input section list for *.abc*. Input sections may appear in the input section lists of multiple output sections qualified with the **maxsize** subcommand.

Unless you are performing a partial link, every section the linker encounters in the object files it reads must appear, either explicitly or implicitly, in an input section list of a **section** command. If the linker encounters a section in an input object file whose name has not been specified as an input section, it will report an error. For this reason, it follows that the linker must read your linker command file(s) before it reads any object files.

The rest of the components of a **section** command are known as *subcommands*. Subcommands modify a particular behavior of the linker with respect to the section being produced by the enclosing **section** command. Described below are the subcommands, the effect they have on the linking process, and the linker's default behavior in the absence of the subcommand. Groups of commands that are mutually exclusive with one another will be described together.

set

set *symbol* = *expr*;
set *symbol* <= *expr*;

The **set** command is used to define new global symbols. It is similar to the **let** command except that the expression, *expr*, is evaluated after the entire linking process has taken place. This means any possible expression is legal in a **set** command. Furthermore, a **set** command creates a *soft definition*, whereas the **let** command creates a *hard definition*. The symbol defined by a **set** command is only placed in the symbol table of the output file if it is actually used; and only if it is not defined elsewhere in your program, i.e., defined as a global symbol in one of the input object files.

The second form of the **set** command works identically to the first except that if an object file contains a weak **export** of *symbol*, the **set** value given in the linker command file will override the object file's definition of *symbol* and the definition of *symbol* in the object file becomes local to the object file.

short

opthdr short {[*order*]} = *expr*

The **opthdr** command allows you to produce a data structure in the output object file called the optional header. The **short** subcommand is used to place a 16 bit value in the optional header in the byte order specified by the optional *order*.

```
opthdr short[0,1] = 1000;
```

The expressions in the **opthdr** command may use symbols that will be defined later in the link process.

string

opthdr string = *string*

The **opthdr** command allows you to produce a data structure in the output object file called the optional header. The string subcommand is used to place a character string in the optional header.

```
opthdr string = "test";
```

text

section *name* text

The **text** subcommand cause an appropriate ICOFF section flag to be set for the section containing the subcommand. The **text** flag is used to indicate a section containing executable. The interpretation of this flag within an ICOFF file is dependent on the program that ultimately reads the file. Introl's debugger, for example, uses the fact that a section only contains data (i.e. was defined with the **data** subcommand) to set breakpoints more efficiently on certain targets.

If these flags are required by an Introl utility, the documentation for the utility in question will say so.

window

window *name* [rom|ram|eeprom|io] [bss] origin *expr* maxsize *expr*

The **window** command is used to associate groups into a defined memory window. The purpose of the window command is to define an area of memory that will have different contents from time to time due to some form of bank switching. The **window** command is only useful with processors with an address space of 64K or less.

The **window** command must have an **origin** and a **maxsize** subcommand. These specify the base and size of the window that will be banked. The **rom**, **ram**, **eeprom,** and **io** device types are optional and are implicitly added to each group associated with a window, as is the optional **bss** attribute.

There can be up to 16 banking windows defined for a program, each with up to 4096 pages to be banked. The linker numbers the windows internally from 0 to 15. The pages are numbered 0 to 4095 by default (the page numbers can be controlled by the group **bank** subcommand). The bottom 16 bits of the address of a symbol in a windowed section is normal. the upper sixteen bits are allocated as follows: the top 4 bits are the window number and the next 12 bits are the bank number.

group *name* window *windowname* [bank *expr*]

The **window** subcommand associates a group with a window for bank switching. The window must be defined before a **group** command refers to it. The optional **bank** subcommand tells the linker to use *expr* as the bank identifier rather than the default internally generated sequential number.This could be used, for example, to make the bank identifiers correspond more closely with the underlying hardware bank switching mechanism. For example the hardware might allow eight banks, each selected by a bit in an i/o port. The bank identifiers for a window like that might be 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, and 0x80. Bank identifiers can be any value between 0 and 4096 (12 bits).

Bank identifiers for all groups in a window must be unique.

Partial linking

The term *partial linking* describes the overall process of building a program using several link steps. Normally, you link all the object files that make up a program using a single run of the linker. When employing partial linking, the final program is still produced from a single run of the linker, but at least some of the input files to the final link have been generated by previous linker runs rather than being generated by the assembler.

Why partial link?

You will have to determine if partial linking will suit your program, but in general, some large programs that have many source files that are kept in different directories or on different systems may be well suited to partial linking. In the case of a program whose source files are grouped into many directories it would make sense, for example, to partially link groups of files from each of the directories.

As an example of partial linking, consider a program that is produced by linking the object files *a.o*, *b.o*, *c.o* and *d.o*. A normal single pass link would just link the four object files and produce the final object file. When partial linking, you might first link the files *a.o* and *b.o* producing *ab.o* then link the files *c.o* and *d.o* producing *cd.o*. Finally, you would link *ab.o* and *cd.o* producing an (almost) identical final result as in the case where partial linking is not used.

How to partially link

The linker normally produces an absolute object file that is not suitable for input to a subsequent linker run. The **−r** command line option tells the linker to perform a partial link which makes it create a relocatable object file rather than an absolute object file. In addition to using the **−r** command line option, you must also use a different linker command file. A normal linker command file would contain, among other things, a **section** command for each section that you want in the output file. Each of these commands would, either explicitly or implicitly, contain an **origin** subcommand indicating where the section should be located in memory. A linker command file for a partial link is generally very simple and contains only a set of **section** commands for each section contained in any of the input files.

For example, consider the following fragment from a linker command file that is normally used to produce an absolute file (i.e. not for a partial link):

```
...
section .code origin 0x1000 = .code;
section .data itemalign 2 = .data
section .ram bss itemalign 2 = .ram;
...
```

These **section** commands tell the linker to place the sections *.code*, *.data* and *.ram* in memory one after another beginning at the address 0x1000. Furthermore, the linker will only place data in the sections in the output file that come from a section in the input file with a corresponding name. Finally, it tells the linker that each chunk of data from both the *.data* and the *.ram* sections (note that the *.ram* section is also given the BSS attribute) taken from the input file(s) should be aligned on an even memory boundary.

If you wanted to perform a partial link given a program that used the linker command file fragment's section commands listed above, you would create a linker command file that just contained the **section** commands and did not have any **origin** subcommands. In this case, the linker command file for partial linking would be as follows:

```
section .code = .code;
section .data itemalign 2 = .data
section .ram bss itemalign 2 = .ram;
```

The final link of the program would be carried out as normal using the original linker command file but specifying the results of previous linker runs as input to the linker rather than using the original object files produced by the assembler.

Linker listing formats

The linker can display three different types of information in its listing output: *section maps*, *symbols tables*, and *file usage*. Using the **−m** option produces a section map. All numeric data is displayed both in decimal and hex. The section map displays each output section used, giving its name, size, origin (if absolute), BSS attribute if the section was defined as BSS, and a list of input sections. For example, the section map might look like this:

```
*** Section Map***
Output Section: '.text', Size 5718 (0x1656)
        Origin: 8 (0x8)
        Input Sections:
        '.text', size 5718 (0x1656)
Output Section: '.heap', Size 4096 (0x1000)
        Origin: 7124 (0x1bd4)
        BSS (non−initialized data)
        Has no input sections:
```

The **−y** option produces a symbol table listing. Each symbol is grouped by input file. The symbol's name, value, and output section are displayed. After all known symbols are displayed, the common, set and undefined symbols are displayed. If a common variable is displayed before it is placed in a section, the size will be displayed instead of its value. An example symbol table might look like:

```
*** Symbol Table ***
In File: '/usr/introl/lib20/start.o'
        .bss            5896 (0x1708), section: 3
        start           8 (0x8), section: 1
In File: 'ali.o20'
        main            198 (0xc6), section: 1
```

If the **−u** option is specified, a file usage map is displayed. Each file used and its type is displayed. If the file described was an archive, a list of members used will be displayed. An example file listing would be:

```
*** Input File(s) Processed ***
Command File: /usr/introl/lib20/c.ld
Object File: /usr/introl/lib20/start.o
Archive File: /usr/introl/lib20/libc.a20
Member(s) Used:
        printf.o20, terminal.o20, strlen.o20, ofmt.o20
```

Warnings and errors

value too large for relocation_type in filename
: This warning is produced when a relocation expression evaluates to a value larger than the relocation record will hold. However, this is only a warning, and the linker will simply truncate the value and continue.

section already has an origin in filename

Each section can only have one origin.

section didn't fit into any predefined output section

The section called section did not fit into any of the output sections which take it as an input. This indicates that all output sections that the input section is assigned to have **maxsize** subcommands.

section does not meet alignment restriction

The section does not meet the alignment restrictions it was given in its section declaration.

section does not meet item alignment restriction

This conflict can only happen when an **origin** subcommand positions code or data to be unaligned.

section in file filename has a different origin than previous section

Two sections with the same name have conflicting origins.

Absolute section in filename follows previous section of non−zero size

An input section of non−zero size has already been placed into an output section where another input section with an origin was attempted to be placed.

Archive filename has no symbol table

The library file filename has no symbol table in it, probably because it contains no object files.

Attempt to make section into BSS section in commandfile

An input section that was not defined as a BSS section was attempted to be placed into an output section defined as a BSS section.

BSS section in file filename follows previous non−BSS section

A BSS input section was going to an output section where previous input sections were not BSS sections.

Bad character in commandfile

An invalid character was found in the linker command file *commandfile*.

Bad operator in expression in command file commandfile

An invalid operator was found in an expression in the linker command file *commandfile*.

Bad section in symbol symbol from command file commandfile

The symbol *symbol* is in an invalid section.

Badly formed option in commandfile

There is a syntax error in the use of an option in the command file *commandfile*.

Can't open filename

The file *filename* could either not be found, or permissions did not allow it to be opened.

Can't open library file filename

The library *filename* could either not be found, or permissions did not allow it to be opened.

Cannot find a section in which to allocate COMMS

A **comm** symbol was found, but no section was declared with the **comms** flag.

Comm Symbol symbol is differently sized in filename

More than one **comm** symbol was found, and they are of at least two different sizes.

Common section in filename follows previous non−Common section

A common input section was going to an output section where previous input sections were not common sections.

Division by zero in commandfile

An expression in *commandfile* generated a division by zero.

Endof(section) has unevaluated origin in commandfile

It is illegal to find the end of a section that has no origin, or one that is not known yet.

Endof(section) not fixed in command file commandfile

It is illegal to find the end of a section that is not (as yet) fixed.

File alignment must be > 0 in commandfile

Alignment value given with **filealign** must be positive.

Illegal octal constant in commandfile

A constant intended to be in octal format was invalid.

Maxsize of section smaller than previous in commandfile

An output section was re−defined having a **maxsize** smaller than previous declarations.

Minsize of section smaller than previous in commandfile
An output section was re−defined having a **minsize** smaller than previous declarations.

Missing ')' in commandfile
commandfile has a syntax error dealing with mismatched parentheses.

Mod by zero in command file commandfile
An expression in *commandfile* tried to do a mod ('%') by zero.

Multiple definition of section section in commandfile
Section *section* was defined more than once in *commandfile*.

Name needed for section directive in commandfile
Syntax error in **section** command in *commandfile*.

Name needed for import directive in commandfile.
The **import** directive requires at least one symbol or an empty list terminated by a semicolon.

Name needed for set directive in commandfile
Syntax error in a **set** command in *commandfile*.

Section section has been used as input to section, which has no MAXSIZE
This warning is displayed when multiple output sections are defined with the same input section. If a previous output section is found without a **maxsize** defined, the other output sections will be ignored.

Semicolon needed after input sections in commandfile
Syntax error −− missing semicolon on a **section** definition.

Setting origin of non−empty section not allowed in commandfile
The linker cannot force the section to start at the given origin, possibly because of an **origin** subcommand. The input section probably had an origin set by a previous linker run or by the **section** assembly directive.

Startof(section) has unevaluated origin in commandfile
It is illegal to find the start of a section that does not as yet have a defined origin.

Startof(section) not fixed in command file commandfile
The start of a section cannot be found if the section does not have a fixed location.

String too long in commandfile
Strings are limited to 2K in length.

Symbol symbol redefined in filename
The symbol *symbol* was defined more than once. The second definition of the symbol came from *filename*.

Symbol symbol used in command file commandfile is not in absolute section
The expression which used *symbol* in *commandfile* required it to be absolute, and it was not.

Syntax error in commandfile, unexpected token token encountered
Syntax error at *token* in commandfile.

Syntax error in expression in commandfile
A syntax error was found in an expression somewhere in *commandfile*.

Undefined section section used in command file commandfile
section was used (e.g. in a **endof()**) but not defined in *commandfile*.

Undefined symbol symbol used in command file commandfile
A symbol was used in an expression, but is not defined in the command file or in any of the object files.

symbol didn't fit into any comms section
A **comm** symbol was found, and no more room was found in the **comms** section.

Internal Errors

These errors indicate a corrupt ICOFF file or an error in the linker program.

anonymous section in file filename is not absolute
bad option option
bad relocation format in filename, section
bad relocation type in filename, section
bad size in relocation type in filename, section
bad symbol table in archive filename
can not open listing file filename for writing
can't reopen filename in pass two
error padding filename, section
error reading line number record in filename
error reading raw data in filename [, section]
error reading relocation record in filename
error sending raw data in filename, section
error writing line number record from filename
error writing relocation record from filename
error writing relocation record from filename
error writing symbol table
file filename is not an archive in pass two
improperly formatted STYP_EXPR in <filename>
ldahread failed in filename in pass two
ldahread failed in archive filename
ldfhread failed in filename in pass two
ldfhread failed in archive filename, module member
ldgetname failed in filename
ldlseek failed in filename pass two
ldrseek failed in filename in pass two
ldshread failed in filename in pass two
ldshread failed in filename
ldsseek failed in filename in pass two
ldtbread failed in filename
lost section from filename in pass two
lost an anonymous absolute section from filename in pass two
magic number different in filename in pass two
magic number mismatch in filename
off by count symbols
off by count undefs
out of memory
problems opening section
too much data found in filename, section

The Utilities

This chapter describes additional utilities that come with Introl−CODE.

i695
: Convert ICOFF files into IEEE695 debug files.

iadr
: Create a listing file containing the program source and the addresses where the compiled code resides after linking.

iar
: Create and modify a library of files.

ibuild
: Build a program (Like *make*).

idbg
: Create Pentica DBG files from an ICOFF file.

idump
: Display the contents of an ICOFF file in human readable form.

ihex
: Convert ICOFF files into several hexadecimal file formats, such as S−records and Introl hex.

ihp
: Convert ICOFF files into HP64000 emulator format.

imerge
: Create a merged C and assembly language source listing.

ipe
: Convert ICOFF files into P&E Microsystems debug files.

isym
: Create a user defined text file contaning symbol information in an ICOFF file.

i695

i695 converts object files from ICOFF to IEEE−695 format.

```
i695 [options] file
```

Command line syntax and options

−k
Display **i695**'s version when run.

−v
Run in verbose mode.

−b bytes
Specify the number of bytes in an address.

−o file
Specify an output file name.

−p processor
Specify the processor.

Official Names

- 68000
- 68008
- 68010
- 68012
- 68020
- 68030
- 68040
- 68HC32
- 68HC16

Additional Introl Names (not HP approved)

- 68332
- 68EC030
- 68EC040
- 68HC05
- 68HC08
- 68HC11
- 68HC12
- 6809
- 6301

−s symbol
Specify the name of the symbol where execution should begin.

iadr

iadr prints the addresses of executable lines, static variables and global variables in a source listing that can be created after a program is linked.

iadr uses information in the ICOFF file created by the linker to produce the listing.

Command line syntax and options

```
iadr [options] objectfile [{sourcefile}]
```

With no source files specified, **iadr** displays all the source files referenced in the object file. An address is displayed if the object file has a corresponding line number entry. If a source file is referenced and cannot be found, **iadr** ignores it. If source files are specified on the command line, **iadr** restricts its output to those source files only.

−a
: Display source files in alphabetical order, sorted by source file name. Normally, the source files are printed in the order they are encountered in the object file.

−c
: Complain about source files that cannot be found. By default, **iadr** will ignore any unspecified source files that cannot be opened. Source files explicitly specified on the command line will always cause an error if they cannot be found.

−i path
: Search for source files in the directory *path*. **iadr** always looks for source files as specified in the object file first, then will prepend each pathname with the directories specified with **−i** options. Multiple **−i** options can be specified on the command line and are searched in the order they are specified.

−k
: Display **iadr**'s version number.

−l length
: Set the output page length to *length* lines.

−n
: Suppress the output of source line numbers in the output listings.

−o filename
: Re−direct listing output to *filename*.

−p
: Disable *pagination*. No page numbers, page headings or listing summaries will be displayed. Instead, a single line containing a source file name will separate different source listings.

−r range
: Display only source files that contain lines with addresses within the *range* specified. The address range is specified with two hexadecimal numbers separated by a dash (−) without spaces. Hexadecimal numbers can be optionally preceded by **0x**. Only the last **−r** option on the command line is effective.

−t
: Use line feeds instead of form feeds to separate pages.

−w width
: Set the output page width to *width* columns.

Output format

By default the listings are paginated. Each page has a heading consisting of the current source file name, the object file name, a date and time, and a page number. The listing is organized into source file listings followed by a listing summary. Each source file line has the following format:

```
AAAAA BBBBBBBB XXXX?
```

- Field A is the source line number
- Field B is the memory address in hexadecimal
- Field X is the source line

For example:

```
iadr a.out
```

produces a paginated listing to the standard output, consisting of all source files that are referenced by *a.out* complete with line numbers and memory addresses.

The command:

```
iadr −r 0x1000−0x2000 a.out
```

restricts the listing to source files whose lines occupy memory in the range of 0x1000 to 0x2000.

The command:

```
iadr a.out file.c
```

restricts the listing to *file.c*.

Error messages and recovery

address range not found
: The address range specified with the **−r** option could not be displayed. Either the object file does not contain line number entries for the specified memory range or the source file could not be found.

bad magic number in filename
: An object file was specified that was corrupted or not in ICOFF format.

bad option option
: The option found on the command line was invalid, or had a syntax error.

can't find source file filename
: The source file named *filename* was not found in any of the paths.

can't open filename
: The object file named *filename* was not found.

can't open output file filename
: The file specified by the **−o** option could not be created.

can't open source file filename
: The source file *filename* could not be opened for reading. This could be a problem with file permissions.

filename must be specified with −o
: The **−o** option was specified, but no filename was given.

ldahread failed
ldlnread failed
ldlseek failed

ldtbread failed
: An internal error occurred when reading the object file. Either the file was not an ICOFF file, or it was corrupted.

line length too short
: Line length must be at least 80 characters.

no source files found
: No source files were located and nothing was printed.

object file did not reference filename
: A source file was specified that the object file did not reference.

out of memory
: **iadr** ran out of memory.

output file defined twice
: Two **−o** options were specified, and only one is allowed.

page length too short
: Page length must be at least 10 lines.

path must be specified with −i
: A **−i** option was specified, but the required path argument was not given.

range option must have two addresses
: The **−r** option requires an address range as an argument in the form:

```
−r 0x100−0x200
```

second address in range option must be larger than first
: Addresses in a **−r** option must be in ascending order.

iar

iar is used to manage *libraries*. A library is a file containing one or more files. CODE uses library files to make commonly used routines, such as the CODE libraries, to be easily accessable at link time. In many of the subsequent examples the **−v** option will be used; this option gives verbose descriptions of the operation of the program. The format of a library file is described in Library file format.

iar can be used to create your own libraries of commonly used routines as well as to modify routines in existing libaries.

Command line syntax and options

```
iar option library [{name}]
```

option is one character from the set **drtpx**, optionally concatenated with one or more characters from the set **vuc**. library is the library file. The *name* arguments are constituent files in the *library* file. The dash ('−') character is optional for **iar** options. The meanings of the option characters are:

−r
: Create a library or replace files in a library.

−d
: Delete files from a library.

−p
: Print the contents of all files or those specified.

−t
: Print the names of all files or those specified.

−u
: When used with the **−r** option only, will replace files in the library only if they are newer.

−v
: When used with any option, will cause verbose output.

−x
: Extract all files or those specified. The library is not changed.

Usage

To create a new library:

```
iar −r mylib.a68 test.o68 test1.o68
```

To replace *test.o68* in the library:

```
iar −r mylib.a68 test.o68
```

To extract *test1.o68* from the library:

```
iar −x mylib.a68 test1.o68
```

Error messages and recovery

[u] valid only with [r]
: The **−u** modifier only applies to the **−r** (replace) command.

ldahread failed
: There was an error reading the archive header from an archive. The file is either not an archive or is corrupted.

ldgetname failed
: There was an error getting the name of a symbol from an ICOFF file. The file is either not in ICOFF format or is corrupted.

ldtbread failed
: There was an error reading a symbol table entry from an ICOFF file. The file is either not in ICOFF format or is corrupted.

one of [drtpx] must be specified
: One of the function specification characters must be given.

only one of [drtpx] allowed
: **iar** must perform one and only one operation for invocation. The operations are delete, replace, table of contents, print and extract and are specified, respectively, by the characters **d**, **r**, **t**, **p** and **x**.

out of memory
: A request to the operating system for memory failed.

can't delete name for renaming
: Under some operating systems (namely MS−DOS), **iar** must delete an existing archive file before renaming its temporary file. This error indicates that the existing archive could not be deleted.

error renaming name to new_name
: **iar** creates new archives in a temporary file and renames the temporary file after all the operations are completed. This error indicates that the temporary file could not be renamed. In this case, the temporary file should still exist. The name of the temporary file should start with the letter **v** and be followed by five or six digits.

name does not exist
: The named file does not exist.

name not in archive format
: The named archive file is not in archive format. Most likely the wrong file was specified or the archive is corrupted.

can't create name
: The named file cannot be created.

can't ldopen name
: An ICOFF file or an archive is corrupted or nonexistent and couldn't be processed.

can't open name
: The named file could not be opened. Most likely it is one of your named files and does not exist.

error reading name
error seeking name
error writing name
: For these three errors, the indicated operation failed. Either the file is corrupted, or your operating system's file system needs checking.

ibuild

ibuild is used to build programs. It is similar to the program **make**. **ibuild** is controlled by a *makefile*. A makefile is a set of rules used by **ibuild** to control how a program is built. If either a file named **makefile** or **Makefile**, in that order, exists in the current directory, **ibuild** will read that file as the makefile. The default makefile names can be overridden with the **−f** option.

Command line syntax and options

```
ibuild option [{variable=value}] [{name}]
```

where *option* is one or more of:

−d
: Print out debugging information.

−f makefile
: Use *makefile* as the makefile.

−i
: Ignore exit status of executed commands.

−k
: Print out **ibuild**'s version number.

−n
: Display commands that **ibuild** would execute but do not actually execute them.

−q
: Do not execute any commands, but exit 0 if the specified targets are up to date, otherwise exit 1.

−r
: Do not use the predefined rules.

−s
: Do not echo commands as they are executed.

−t
: Update the modification time of files that need to be built without building them so they appear up to date.

variable=value
: Set a variable to a value.

Usage

There are four types of lines in a makefile: file dependency specifications, commands, variable assignments, and comments.

Lines may be continued from one line to the next by ending them with a backslash (`\').

File dependency specifications

Dependency lines consist of one or more *targets*, an *operator*, and zero or more *sources*. A dependency line creates a relationship where the targets *depend* on the sources and are usually created from them.There are two operators, `:' and `::'. The `:' operator says that the target is out of date if its modification time is less than those of any of its sources. A target defined with the `:' operator may only be listed as a target in one dependency specification with commands following it. The `::' operator is like the `:' operator except that the target may be specified in more than one dependancy specification and each may have commands following

them.

Commands

Every line following a dependency specification that begins with a tab character is a command associated with the targets. If any of the targets of a dependency specification is older than any of the sources of the specification, then the commands associated to the dependency specification are executed, presumably to create the desired target or targets.

If the first non−blank characters of a command line may be `@' or `−' with the following meanings. A line preceeded by a `@' character is not echoed before it is executed. A `−' causes any command that returns an error status to have that status be ignored: normally an error status will terminate **ibuild**. Both the `@' and `−' characters may preceed a command line.

Variable Assignments

Variables assignments have the form *name = value*. This assigns the string *value* to the variable *name*.Variables are used in *substitutions*, described below.

Substitutions

Substitutions occur when a dollar sign (`$') is followed by a single character not equal to `{' or '(' or one or more characters bracketed by `{' and `}' or `(' and `)'. Substitutions will done as follows using the character or characters following the dollar sign (minus the brackets) as the *name*:

1. if *name* is `$', a `$' is substituted, or
2. if *name* is `<', the name of the source from which the target is to be transformed is substituted, or
3. if *name* is `?', the names of the sources for which this target that were out of date are substituted, or
4. if *name* is `*', basename of the target, minus any extension or preceeding directory path is substituted, or
5. if *name* is `@', the name of the target is substituted, or
6. if *name* was defined on the ibuild command line, the value of that name is substituted, or
7. if *name* was defined in the makefile, the value of that name is substituted, or
8. if name is defined as an environment variable, the value of that name is substituted,
9. otherwise the substitution results in an empty string.

Substutions can occur in dependency specifications, commands, and as the value of a variable assignment. Susbstitutions in dependency specifications are done when the specification line is read in. Substitutions in command lines are made when the command is executed. Substitutions in the value of a variable assignment occur when the variable is used; the value is not expanded before assignment.

Comments

Comments begin with a hash character (`#') and continue until the end of the line. Hash characters are ignored in command lines.

idbg

idbg converts ICOFF files to Pentica DBG file format. This format is used by the Pentica source level debugger, **MIMEview**. To convert a compiled and linked program named *example* to DBG format use the following command line:

```
idbg example
```

This command creates all the files required for DBG format: *example.dbg*, *example.syc*, and *example.s2*.

idump

idump extracts information from an ICOFF object file. You will not normally need to use this program; however, if you contact Introl technical support, you may be asked to use it. For many of the following examples the **−v** option will also be included, this option will display flags normally displayed as hexadecimal values in a mnemonic form. White−space separating an option and its modifier are optional.

Command line syntax and options

```
idump [options] {filename}
```

The filenames given to **idump** can be object files or libraries.

−a
: Print the header of a library. If the argument is an ICOFF file, the option is ignored.

−c
: Print the string table.

−f
: Print the file header.

−h
: Print the section headers.

−i
: *Print the sizes sections, grouped by section type. The sizes of all sections defined with the **data**, **text**, and **bss** section flags, and the total size of all sections are printed.*

−l
: Print the source line number records.

−m name
: Print a listing containing the address of the beginning of the section named *name* in each source file.

−o
: Print the optional header.

−r
: Print the relocation information.

−s
: Print a hexadecimal dump of the raw data in all sections or those specified.

−t
: Print the symbol table.

−w string
: Print a string from the optional header, when the optional header has a format of:

```
identifer=value[{:identifier=value}]
```

If *string* is specified, it is used as *identifier*. If a dash (−) is specified instead of *string*, the entire optional header is printed.

−d start_number
+d end_number
: These options will restrict the sections printed by **idump**. If the **−d** option is specified, *start_number* is the first section printed. If the **+d** option is specified, *end_number* is the last section printed.

−n name
: Print information for symbol *name* only. This option only applies to the **−h**, **−s**, **−r**, **−l**, and **−t** command line options.

−p

Suppress the printing of the output headers which explain the output.

−t start_number

+t end_number

These options will restrict the symbol table entries printed by **idump**. If the **−t** option is specified, *start_number* is the first symbol table entry printed. If the **+t** option is specified, *end_number* is the last smbol table entry printed.

−v

Print the flags and time−stamps in mnemonic form instead of the default hexadecimal.

Error messages and recovery

bad magic number

The ICOFF file header magic number was invalid.

bad option

The option or option argument was invalid.

can't open filename

An error was returned from the operating system during an attempt to open the input file.

ldahread

ldohseek failed

ldrseek failed

ldsseek failed

ldlseek failed

ldahread failed

ldtbread failed

ldgetname failed

The object file is corrupt in some way.

ihex

ihex converts ICOFF files to one of several different hex file formats:

- Motorola S records
- Intel hex records
- Tektronix hex records
- Extended Tektronix hex records

Command line syntax and options

```
ihex [options] filename
```

−d
: Use the device origin rather than the actual memory address to generate hex files. This is used in bank−switched systems with large memory devices.

−e flag
: Set the *end−of−line sequence*, where *flag* can be any combination of the following characters: **n**, **r**, or **c**, where **n** represents a newline (line feed, 0x0A), **r** represents a return (0x0D), and **c** represents a return and a new line, in that order. Any combination of the above letters may be used.

−f filename
: Read the command file *filename*. A command file contains a list of options and/or output specifications. Command line options can be placed, one per line, in the file. A specification table may follow the list of options. The **−f** option cannot be used inside a command file.

−g flag[1|2|3]
: Set the output file format. *flag* may be one of **m**, **i**, **t**, or **x**, where **m** is Motorola S−records, **i** is Intel hex, **t** is Tektronix hex, and **x** is Extended Tektronix hex. The default format is Motorola S−records. If *flag* is **m** (Motorola S−records) then a single digit can be specified indicating the smallest data record type to use. Normally, **ihex** will create the smallest data record (S1, S2, or S3) that can contain the address of the hex bytes generated. A digit following the **−gm** option will cause **ihex** to create data records with larger than the minimum size, e.g. **−gm2** will cause **ihex** to create data records with only S2 (and S3, if needed) records. The default is **1**, meaning S1 records will be generated for data between 0x0 and 0xFFFF, S2 records will will be generated for data between 0x10000 and 0xFFFFFF, and S3 records will be generated for data between 0x1FFFFFF and 0xFFFFFFFF.

−h
: When used in addition to the **−gx** option will produce additional symbols for source lines and file names, used by Huntsville Microsystems debugging software and emulators.

−k
: Print the program name and version number.

−l number
: Set the number of bytes in a record to number from 1 to 256. The default is 30 bytes.

−o filename
: Set the *base file name* of the output file(s). The form of an output file name is *filename.number* where *filename* is either the name of the first input file (less any characters starting with the first period in its name), or the name specified by this option, and *number* is a number starting at **0** for the first output file and incrementing by one for each additional file. For extended Tektronix hex output, an additional file called *filename.smb* is used if there is more than one output file. This additional file contains the symbol table.

−r
: **ihex** normally ignores relocatable sections. They will be converted if **−r** is specified.

−s symbol

Specify the entry point symbol for use in the termination record of the output file. Note: termination records are only created when only one output file is generated.

−u

ihex normally sorts sections by address; the **−u** causes ihex to process sections in the order encountered in the ICOFF file.

−v

The **ihex** program will print out all options specified, a table containing the output specification, as well as a progress report.

−z

Causes **ihex** to subtract an offset from the address of each output record equal to the base address described in the specification table.

Command file syntax

Command files may contain command line options, one per line, and a specification table. Comments are introduced by an asterisk (*) or an octothorpe (#) on one input line and are ignored by **ihex**.

Specification table

Each line in the table directs **ihex** to produce a single file, or a group of related files. The lines have the following format:

```
start_addr[−end_addr] device[{,device}] [section{,section}]
```

start_addr is the starting address of the output file(s) and *end_addr* is an optional ending address of the file(s). Addresses are in decimal unless preceded by **0x** in which case they are hexadecimal.

After the addresses, a device list is required. A *device* can be one of the following:

```
size[k]
size[k]x2
size[k]x4
ram
```

size is the maximum number of bytes placed in the output file specified by the specification line. If *size* is followed by an optional **k** then the maximum size is assumed to be 1024 times *size*. If **x2** follows the *size*, **ihex** produces two output files split into even and odd bytes. If **x4** follows the *size*, **ihex** produces four output files.

The **ram** device behaves like a size which is equal to that of the address range in the first field of the command file line.

If a plus (+) appears at the end of a device list, auto−repeat mode is enabled. This mode automatically fills the remaining memory with devices identical to the last device in the device list. You must specify an ending address and have an address range that the devices fit into evenly.

Sections

The optional comma separated input section list on the command file line contains the ICOFF section names of input sections for this address range. A section list causes **ihex** to place data from the named sections in the specified address range into that output file or files. Nameless sections must have a corresponding command file line with no sections specified. Do not use a section list unless you have more than one section at a given

address.

This feature was designed to segregate data from different sections located at the same address into different files for paged systems. We recommend that the **copiedfrom** section modifier be used in a linker command file for this purpose.

Usage

The command file controls the behavior of **ihex**. If no command file is specified, **ihex** behaves as if it has a command file of:

```
0−0xffffffff            4194304k
```

ihex will produce a single output file with all data from all sections in a 32−bit address space. This means that the simplest **ihex** command

```
ihex file
```

will convert the ICOFF file *file* to Motorola S records (the default, see the −g option) and place the output in a single file called *file*.0.

Given a program linked with the following directives in the linker command file:

```
section .rom1a origin 0xc000 = .text, .const, .strings;
section .rom1b = ;
section .bss bss origin 0x100 = .bss;
section .data copiedfrom .rom1b = .data;
```

and a target system containing an 8k−byte EPROM at 0xc000 and an 8k−byte EPROM at 0xe000, the following command file would cause **ihex** to generate one output file for each EPROM:

```
0xc000−0xdfff 8k
0xe000−0xffff 8k
```

This command file may also be written as:

```
0xc000−0xffff 8k, 8k
```

meaning that there are two 8k devices in the specified address range.

The command file:

```
0xc000−0xffff 8kx2
```

causes **ihex** to produce two files; the first has a **.0** extension, containing the data from even addresses, and the second has a **.1** extension, containing the data from odd addresses.

Error messages and recovery

bad EOL style, must be an n,r or c
: The parameter given to the **−e** option was incorrect.

bad memory specification
: An incorrect character was found in a memory specification.

can't open control file
: The file specified with the **−f** option could not be opened.

can't open input file
: An input file specified on the command line could not be opened.

device repeat/split must be less than 9
: The number given for device splitting must be between one and nine.

device repeat/split must have a digit
: After the ``**x**" character for a device specification, a digit is required.

failure opening output file
: An error was encountered when opening an output file.

filename must be specified with −f
: A filename is required as a parameter to the **−f** option.

filename must be specified with −o
: A filename is required as a parameter to the **−o** option.

illegal output type, must be an m,i,t or x
: The parameter for the **−g** option was incorrect. Use an **m**, **i**, **t** or **x** .

input data out of range for output files
: None of the specification lines have a suitable address range or the name of the section containing the data did not match any of the names in the section lists of the specification lines that had suitable address ranges.

input file can't be an archive
: Input ICOFF files cannot be a library.

ldgetname failed
ldshread failure
ldsseek failure
ldtbread failed
: The file is corrupted or is not compatible with ICOFF.

memory range conflicts with device size(s)
: The sum of the device sizes on a specification line must match the size of its address range.

only one level of control file is allowed
: A control file cannot contain another **−f** option.

out of memory
: Memory allocation of internal tables, memory ran out. Try a simpler output specification.

output device (RAM,ROM) required
: An incorrect output specification was encountered. All output specification lines must have a device.

record width out of range
: The parameter given for a **−l** option is too large for this style of output.

syntax error, ending address expected
: After a dash (−) character in an output specification line, an ending address was expected.

syntax error, unrecognized command line options
: A line in a command file contain a character or set of characters unrecognized by **ihex**.

unknown EPROM type
: The EPROM found in an output specification line was not recognized by **ihex**.

unknown output device
: The output device found in an output specification line was not recognized by **ihex**.

unknown storage class found
: An unknown storage class was found in the input file. The input file is either corrupted or is not compatible with ICOFF.

you must have a full memory range for +
: In order to use the automatic device repeat feature you must give a complete memory range.

ihp

ihp converts ICOFF files to HP64000 format. Three types of files can be generated:

Absolute files
: contain raw program code and data and have a filename extension of **.X**

Linker symbol files
: contain a symbol table, addresses of the code and data for each linker input object file, and a memory map of the program. These files have a filename extension of **.L**

Assembler symbol files
: contain the pertinent addresses regarding functions and procedures. There is one assembler symbol file created for each linker input object file. The files have an extension of **.A**.

Command line syntax and options

```
ihp [options] filename
```

−a
: Suppress creation of .A files.

−l
: Suppress creation of .L files.

−x
: Suppress creation of .X files.

−g number
: Set processor word size. number can be either 8 or 16. Use 8 for the 8−bit processors in the 6800 family, and 16 for any 16− or 32−bit processors in the 68000 or 68300 family. If no −g option is used, **ihp** uses a word size of 8.

Building a program for the HP64000 series emulators

The 64000 emulators expect to see only one code and data section in an object file. **ihp**, therefore, generates a warning message if it finds more than one of each of these sections. **ihp** determines the types of the section by looking at the section flags in the object file. These flags are set when linking the program. For example:

```
section .foo text origin 0x1000 = .text;
section .bar data origin 0x8000 = .data, .strings;
```

The **text** subcommand marks section *.foo* as containing code, and the **data** subcommand marks section *.bar* as a data section.

The program must be compiled with the −g option to enable source level debugging.

imerge

imerge creates a mixed listing combining C source and the assembly language produced by its compilation.

Command line syntax and options

```
imerge [options] sourcefile assemblyfile
```

sourcefile
: Name of C source file

assemblyfile
: Name of the assembler source file produced by the compiler when *sourcefile* is compiled using the **−r** option, or it can be the name of an assembler listing file produced by compiling *sourcefile* with the −l option.

−o filename
: Write output to *filename*, instead of standard output.

Usage

Given a C source file, *sieve.c*, the command:

```
cc68 −r sieve.c
```

will produce *sieve.s68*. To merge the C and assembly files:

```
imerge sieve.c sieve.s68
```

If *sieve.c* were compiled with the command:

```
cc68 −r −l sieve.c
```

the compiler will produce *sieve.lst*. To merge the C and listing files:

```
imerge sieve.c sieve.lst
```

imerge will not necessarily arrange the assembler code in the way you might expect since it may associate disjoint pieces of assembler code with one source line.

Error messages and recovery

bad option option
: An unknown option was encountered on the command line.

can't open assembly file filename
can't open output file filename
can't open source file filename
filename must be specified with −o
: A file is required as an argument when specifying the **−o** option.

out of memory
: The **imerge** program ran out of memory.

output file defined twice
: An output file can only be specified once.

two source file names must be specified
: Both the C source program and the assembly file must be specified.

ipe

ipe converts object files from ICOFF to P&E map files. **ipe** creates two output files by default. The first, with the extension **.s1** contains Motorola S records. The second output file contains the proprietary P&E format symbol table file and has the extension **.map**.

Command line syntax and options

```
ipe [options] file
```

The files produced by **ipe** have the names *file*.s1 and *file*.map.

−k
: Display **ipe**'s version number when run.

−d
: Turn off the generation of the **.map** file.

−s
: Turn off the generation of the **.s1** file.

−v
: Run in verbose mode.

−m
: *Limit number of source files retained to 64. This option was added to get around limitations in some P&E format file consumers that only enable them to handle 64 source files. Any additional source files referenced in the ICOFF file are ignored.*

isym

isym reads an ICOFF file and creates symbol table listings in a user defined format. **isym** is useful for making symbol table files that can be read by emulators that are unable to read a statndard file format.

Command line syntax and options

```
isym [options] object_file[{,object_file}]
```

−a
Sort the symbols in alphabetical order. By default, the symbols are unsorted.

−c filename
Use the command file *filename* instead of **default.sym**.

−d
Inhibit the reading of **default.sym**.

−e string
Append *string* to the format string used for the epilogue.

−k
Display **isym**'s name and version.

−l
Treat all symbol names as lower case. This option affects both sort order and the case of the displayed symbol name.

−m string
Append *string* to the format string used for symbolic output.

−n
Sort numerically by symbol value.

−o name
Specify the output file name. Without this option, the output is displayed on the standard output.

−p string
Append *string* to the format string used for the prologue.

−r
Sort in descending or reverse order.

−s
Display static symbols as well as external symbols.

−u
Treat all symbol names as upper case. This option affects both sort order and the case of the displayed symbol name.

−x symbol
Restrict the output to the named symbol. Multiple **−x** options are allowed and will restrict the output to named symbols.

Command file

The command line is scanned from left to right. If a command file is specified, the options inside of the command file will be read after the command line options. By default **isym** looks for a command file named **default.sym**. The **−c** option can be used to specify a different command file.

Any line that starts with a single dash (−) is parsed as a command line option. If two dashes (−−) are found, the line is considered to be a symbol format line starting with one dash. Any other lines in the command file are considered format strings.

Format strings

Format strings, the *prologue string* and the *epilogue string*, are used to define the output format. The prologue string is displayed at the beginning of the output and the epilogue string is displayed at the end of the output. The format strings can be specified both on the command line and in a command file.

You can use the **−m**, **−p** and **−e** options to specify the symbol format, prologue format and the epilogue format. By default, these strings are blank and produce no output.

isym applies each format string to each global symbol in the object file, unless **−s** is specified in which case each format string will be applied to each static symbol as well. As **isym** processes a format string, it prints the string's contents until an escape sequence is encountered, which **isym** expands as described below.

%n[([−][number][case])]
: Prints the symbol's name. If an optional dash (−) is specified, the symbol's name will be right justified instead of left justified. If an optional *number* is specified, the width of the symbol name's field will be set to *number* characters. If an optional *case* is specified, one of **l** or **u**, the symbol's name will be printed in lower case or upper case, respectively.

%v[([−][[0]number][format])]
: Print the symbol's value as an integer. If an optional dash (−) is specified, the symbol's value will be right justified instead of left justified. If an optional zero (**0**) is specified, the symbol's value will be padded with leading zeroes. If an optional *number* is specified, the width of the symbol's value field will be *number* characters wide. If an optional *format* is specified, one of **H**, **h**, **b**, **d**, or **o**, the value will be printed in upper case hexadecimal, lower case hexadecimal, binary, decimal or octal format, respectively.

%c[([−][[0]number][format])]
: Print the symbol's storage class as an integer. If an optional dash (−) is specified, the symbol's storage class will be right justified instead of left justified. If an optional zero (**0**) is specified, the symbol's storage class will be padded with leading zeroes. If an optional *number* is specified, the width of the symbol's storage class field will be *number* characters wide. If an optional *format* is specified, one of **H**, **h**, **b**, **d**, or **o**, the storage class will be printed in upper case hexadecimal, lower case hexadecimal, binary, decimal or octal format, respectively.

%%
: Print a percent sign.

r
: Print a carriage return.

l
: Print a line feed.

n
: Print the host system's end of line sequence.

\\
: Print a backslash.

Usage

Given the following **isym** command file *cmd.sym*:

```
−p BEGIN\n
−e END\n
−n
−r
```

```
−m The value of %n is %v\n
```

In the example command file, the **−p** option defines the prologue string, the **−e** option defines the epilogue string, the **−n** option specifies to sort numerically by symbol value rather than by symbol name, the **−r** option specifies to sort in reverse (descending) order, and the **−m** option specifies the format string to use for each symbol.

Using ctour as an example program, the following command line:

```
isym −c cmd.sym ctour
```

produces output similar to:

```
BEGIN
The value of _initend is 00057582
The value of _initstart is 00057560
The value of main is 00057487
The value of subr is 00057471
The value of _returnpoi is 00057464
The value of _debugtrap is 00057423
...
The value of H11PORTB is 00004100
The value of H11PORTC is 00004099
The value of H11PIOC is 00004098
The value of H11PORTA is 00004096
The value of H11RAMREG is 00000001
END
```

Error messages and recovery

a string must be specified with −g
: The argument for the **−g** was missing.

bad magic number in filename
: The input object file did not have a legal magic number.

bad option option in command file
: An unknown command line option was encountered in a command file.

bad option option
: An unknown command line option was encountered.

can't open filename
: An operating system error occurred when *filename* was attempted to be opened.

can't open command file filename
: An operating system error occurred when *filename* was attempted to be opened.

can't open output file filename
: An operating system error occurred when *filename* was attempted to be opened.

file name must be specified with −c
: The argument for a **−c** option was missing.

file name must be specified with −o
: The argument for a **−o** option was missing.

illegal format string character character
: An unknown format command character was found.

ldahread failed
ldtbread failed
: For either of the above two errors, an error occurred when reading the input object file.

no symbol table format specified

No command file was specified and **default.sym** did not exist.

out of memory for symbol table!

isym ran out of memory when reading in the symbol table.

output file defined twice

The output file was defined in at least two command line options.

File Formats

This chapter tells about the file formats used and produced by CODE.

Libraries
: How library archives structured.

Motorola S records
: A very common hexadecimal format commonly used in conjunction with Motorola microcontrollers.

Intel hex
: Intel's equivalent to S records, need by some EPROM burners.

Tektronix hex
: Yet another hex format, this time by Tektronix.

Extended Tektronix hex
: Not being content with a single format, Tektronix developed their extended hex format which also contains symbol information. (Does anyone still use this?)

Introl Common Object Files
: Description of the Introl object file format, ICOFF.

Libraries

Each library begins with this eight character magic string:

```
!<arch>\n
```

The character sequence \n represents an ASCII newline (line feed, 0x0A) character. Libraries which contain ICOFF object files will also include an archive symbol table. This symbol table is used by the Introl Linker **ild** to determine which members must be loaded during the linking process. The archive symbol table (if it exists) is always the first file in the library (but is never listed) and is automatically created and/or updated by **iar**, the Introl library manager.

Following the magic string (and the symbol table, if one exists) are the library *members*. Each file member is preceded by a file member header which has the following format:

Name	Bytes	Description
ar_name	0−15	file member name
ar_date	16−27	file member date
ar_uid	28−33	file member user identification
ar_gid	34−39	file member group identification
ar_mode	40−47	file member permission bits
ar_size	48−57	file member size
ar_fmag	58−59	file member magic string "`\n"

All information in the file member headers is in printable ASCII. The numeric information contained in the headers is stored as decimal numbers (except for **ar_mode** which is octal). Thus, if the library contains printable files, the library itself is printable.

The **ar_name** field is blank−padded. The **ar_date** field is the modification date of the file at the time of its insertion into the library. Libraries can be moved from host system to host system as long as the portable library command **iar** is used. If the name of the file in the archive is longer than 16 characters, the **ar_name** field is replaced by the string **#1/*size***, where ***size*** is the length of the file name. The file name for long file member names immediately follows the file member header, and the file member size is adjusted to include the size of the file name.

Each library file member begins on an even byte boundary; a newline is inserted between files if necessary. Nevertheless, the size given reflects the actual size of the file exclusive of padding.

If the archive symbol table exists, the first file member in the library has a zero length name (i.e., **ar_name**[0] = `/'). The contents of this file member are as follows:

- The number of symbols. Length: 4 bytes.
- The array of offsets into the library file. Length: 4 bytes * (number of symbols).
- The name string table. Length: **ar_size** − (4 bytes * (number of symbols + 1)). The string table contains exactly as many null terminated strings as there are elements in the offsets array. Each offset from the array is associated with the corresponding name from the string table (in order). The names

in the string table are all the defined global symbols found in the ICOFF files in the library. Each offset is the location of the library header for the associated symbol.

The offsets described above are kept in machine independent forward word order, high order byte to low order byte.

Introl Common Object Files

Both the Introl Assembler and the Introl Linker (**ild**) generate files in this format. ICOFF supports user−defined sections and contains extensive information for source level debugging. An ICOFF object file contains the following components in order:

File Header
: The section of an ICOFF file that describes the file's contents .

Optional Header
: An option section that can contain user generated information.

Section Headers
: One or more section headers define the program code and data contained in the ICOFF file.

Relocation Information
: Information telling the linker how to reposition the ICOFF file in memory.

Line Number Records
: Information about the source files used to create the ICOFF file.

Symbol Table
: Symbol and type information from the original source files.

String Table
: The character strings representing all the symbols in the ICOFF file.

The following type names are used in this document to describe the elements of each part of an ICOFF file.

Type names	Size
char	eight bit signed integer value
short	16bit signed integer value
unsigned short	16 bit unsigned integer value
long int	32 bit signed integer value
unsigned long	32 bit unsigned integer value

File Header

The file header contains the 20 bytes of information as shown below. The last two bytes are flags that are used by **ild** and object file utilities.

Bytes	Type	Name	Description
0−1	unsigned short	f_magic	magic number
2−3	unsigned short	f_nscns	number of section headers (equals the number of sections)
4−7	long int	f_timedat	time and date stamp indicating when the file was created equal to the number of elapsed seconds since 00:00:00 GMT, January 1,1970
8−11	long int	f_symptr	file pointer containing the starting address of the symbol table
12−15	long int	f_nsyms	number of entries in the symbol table
16−17	unsigned short	f_opthdr	number of bytes in the optional header
18−19	unsigned short	f_flags	(see table for a list of file header flags)

The size of optional header information (**f_opthdr**) is used by all referencing programs that seek to the beginning of the section header table. This enables the same utility programs to work correctly on files targeted for different systems.

Magic number

The magic number represented by **f_magic** in the file header identifies the file as an ICOFF object file. The value is always 601 (octal). The magic number can be changed through the use of the **magic** directive in the linker.

File header flags

The last 2 bytes of the file header are flags that describe the type of object file. The currently defined flags are given below:

Mnemonic	Flag	Meaning
F_RELFLG	0x0001	relocation information stripped from file
F_EXEC	0x0002	file is executable (no unresolved external references)
F_LNNO	0x0004	line numbers stripped from file
F_LSYMS	0x0008	local symbols stripped from file
F_AR16WR	0x0080	16 bits, reverse bytes
F_AR32WR	0x0100	32 bits, forward bytes

F_NEWSYM	0x1000	new version of ICOFF symbol table (always set)

Optional Header

The ICOFF format allows user defined information to be placed into the file. The file header variable called **f_opthdr** is used to specify the size of this information. The linker **opthdr** directive can be used to place information into this area.

Section Headers

Object files often have tables of section headers to specify the layout of data within each file. The section header table consists of one entry for each section in the file. Section headers are followed by the appropriate number of bytes of data as shown in the table. The raw data for each section must begin on a full word boundary in the file.

The information in each section header entry is described below.

Bytes	Type	Name	Description
0−7	char	s_name	eight character null padded section name
8−11	long int	s_paddr	physical address of the section
12−15	long int	s_vaddr	virtual address of the section
16−19	long int	s_size	section size in bytes
20−23	long int	s_scnptr	file pointer to raw data
24−27	long int	s_relptr	file pointer to relocation entries
28−31	long int	s_lnnoptr	file pointer to line number entries
32−33	unsigned short	s_nreloc	number of relocation entries
34−35	unsigned short	s_nlnno	number of line number entries
36−39	unsigned long int	s_flags	(see table for a list of section header flags)

File pointers are byte offsets from the beginning of the file that can be used to locate the start of data, relocation, or line number entries for the section. Section names longer than eight characters are supported. When the section name is longer than eight characters, the **s_name** field is replaced by the string ***#index***, where ***index*** is the symbol table index of an **IC_SEC** symbol that describes the section and has the full long name of the section.

Section header flags

The table below contains a list of section header flags. These flags are stored in the **s_flags** field of the section header. The one deviation from the normal rule in the section header table is the entry for uninitialized data in a **STYP_BSS** section. A **STYP_BSS** section has a size and symbols that refer to it, and symbols that are defined in it. At the same time, a **STYP_BSS** section has no relocation entries, and no data entries. In this case, the number of relocation records and data file pointers in a **STYP_BSS** section header are 0.

Mnemonic	Flag	Meaning
STYP_REG	0x0000	regular section (allocated, relocated, loaded)
STYP_DSECT	0x0001	dummy section (not allocated, relocated, not loaded)
STYP_NOLOAD	0x0002	noload section (allocated, relocated, not loaded)
STYP_GROUP	0x0004	group section
STYP_ICODE	0x0008	icode section (allocated, relocated, loaded)
STYP_GROUPEND	0x0010	last section in a group
STYP_TEXT	0x0020	executable text section
STYP_DATA	0x0040	initialized data
STYP_BSS	0x0080	uninitialized data
STYP_DEVICE	0x0F00	Device type: STYP_RAM 0x0000 STYP_ROM 0x0100 STYP_EEPROM 0x0200 STYP_IO 0x0300
STYP_BANK	0x1000	section is banked
STYP_COMM	0x2000	overlay section
STYP_ABS	0x4000	section located at absolute address
STYP_EXPR	0x8000	section contains link time expression information
STYP_BANKNM	0x0FFF0000	bank number if STYP_BANK
STYP_WINDOW	0xF0000000	window bank is associated with, if STYP_BANK

Relocation Information

Object files have one relocation entry for each relocatable reference. The relocation information consists of entries with the format described below.

Bytes	Type	Name	Description
0−3	long int	v_addr	virtual address of reference
4−7	long int	r_symndx	symbol table index
8−9	unsigned short	r_type	relocation type

The first 4 bytes of the entry are the virtual address of the datum to which this entry applies. The next field is the index, counted from 0, of the symbol table entry that is being referenced. The type field indicates the type of relocation to be applied.

As the linker reads each input section and performs relocation, the relocation entries are read. They direct how references found within the input section are treated.

The currently recognized relocation types are given in the table below.

Mnemonic	Flag	Meaning
R_SIZE	0x000f	relocation size mask
R_BYTE	0x0001	eight bit relocation
R_WORD	0x0002	16 bit relocation
R_LONG	0x0003	32 bit relocation
R_12BIT	0x0004	lower 12 bits of 16 bit word
R_24BIT	0x0005	lower 24 bits of 32 bit word
R_VAR	0x0006	variable sized relocation
R_24TBIT	0x0007	lower 24 bits of 32 bit word for 68HC12 CALL
R_RELTO	0x00f0	relocation base mask
R_ABS	0x0000	absolute relocation
R_PCREL	0x0010	PC relative relocation
R_FORMAT	0x0f00	relocation item data format mask
R_FWD	0x0000	forward byte order
R_REV	0x0100	reversed byte order
R_DISP	0x0200	32000 displacement
R_PROC	0x0300	32000 procedure descriptor
R_TYPE	0x7000	type of relocated value mask
R_SIGNED	0x1000	signed value
R_UNSIGNED	0x2000	unsigned value
R_SORU	0x3000	signed and unsigned value

Line Number Records

Line numbers can be used by source level debuggers to correlate points between an object file and a source file. All line numbers in a section are grouped by source file as shown below.

Symbol index of source file name 0	0
Physical address	line number
Physical address	line number
. . .	. . .
Symbol index	0
Physical address	line number
Physical address	line number

The first entry in a source file grouping is indicated by a value of zero in the **l_lnno** field and has, in place of the physical address, an index into the symbol table for the entry containing the file name. Subsequent entries have actual line numbers and the physical addresses corresponding to the line numbers.

Symbol Table

Organization
: The order of symbols in an ICOFF file is very important.

Common Information
: Information common to all symbol types.

Object Symbols
: Object symbols represent objects defined in a program.

Type Symbols
: Type symbols represent types defined in a program.

Special Symbols
: Special symbols represent miscellaneous items defined in or by a program.

Organization

Because of symbolic debugging requirements, the order of symbols in the symbol table is very important. Symbols appear in the sequence shown below.

File name 1
Function 1
Local symbols for function 1
Function N
Local symbols for function N
...
File name 2
Function N+1
Local symbols for function N+1
Function N+X
Local symbols for function N+X
Undefined global symbols

The entry for each symbol is a structure that holds the symbol's value, type, and other information.

Common Information

Symbol table entries are used for representing *objects*, *types*, and *special symbols*. Objects have actual physical addresses and types are used to describe those objects. Special symbols are offsets, registers, filenames and other symbols which do not fall into the object or type category. The *storage class* of a symbol determines its type.

Each symbol can have a name or be anonymous if no name is present. Symbol names are kept in the string table and offsets into that table are used to reference a symbol name These offsets are called *string indices*. A string index of zero represents an anonymous name.

Symbol indices are used in symbol table entries to refer to other symbols. A symbol index is a number in the range of zero to one less than the number of symbols.

Each symbol table entry contains 18 bytes of information.

Common symbol table fields

The following sections describe fields which are common to all symbol table entries.

Each symbol table entry has a class kept in a field called **sy_class**. Classes are used to classify the type of a symbol. A symbol's class will determine what information is kept in its symbol table entry. Classes are listed in the table below.

Mnemonic	Value	Class
IC_EXT	1	global symbol
IC_STAT	2	non−global symbol
IC_LABEL	3	label
IC_FEND	4	physical end of function
IC_FENTRY	5	function entry point
IC_FEXIT	6	function exit point
IC_BBEGIN	7	block begin
IC_BEND	8	block end
IC_SEC	9	section size symbol
IC_SIMPLE	20	simple type
IC_SUBR	21	subrange type
IC_ENUM	22	enumeration type
IC_ARRAY	23	array of . . .
IC_POINTER	24	pointer to . . .
IC_TAG	25	tag type
IC_SETOF	26	set of
IC_TYPE	27	type of
IC_FUNC	28	function or procedure
IC_RECORD	29	record type
IC_UNION	30	union type
IC_CARRAY	31	C array type
IC_LOCAL	50	local variable

IC_PARAM	51	parameter
IC_MEMBER	52	record member
IC_FIELD	53	bit field
IC_LITERAL	54	enumeration literal
IC_FILE	55	file name

The **sy_flags** field of a symbol table entry contains information about symbols that may vary from symbol class to symbol class. The **sy_flags** field is contained in one byte in a symbol table entry. The usage of the **sy_flags** field will be described for each symbol class below.

Mnemonic	Value	Meaning
SF_NAME	0x80	used internally for all symbols
SF_BASIC	0x7F	symbol's atomic type (certain type symbols, none of the flags below apply)
SF_REG	0x10	register variable
SF_VAR	0x20	VAR parameter
SF_ALIAS	0x40	ALIAS parameter
SF_FUNC	0x10	function beginning symbol
SF_WEAK	0x20	IC_EXT is a weak definition
SF_ICODE	0x40	IC_EXT is referenced only from ICODE sections
SF_SPEC	0x01	IC_EXT, IC_STAT is a volatile variable or inline function
SF_INT	0x02	IC_EXT, IC_STAT is an interrupt handling function
SF_PROT	0x04	IC_EXT, IC_STAT is an interrupt protected function

Object Symbols

Object symbols can represent functions, procedures, variables, labels or any program construct that has a physical address. The section number field **sy_scnum** is only used by object symbols and all object symbols have a section number field. The section number describes in which section an object is defined.

A section number of −1 indicates that the symbol has a value but this value is not necessarily an address.

A section number of zero indicates a symbol that is not defined in the current file. This type of symbol is referred to as *imported*. There are two types of imported symbols. The first, and most common, occurs when the imported symbol has a value (in the **sy_value** field) of zero. This type of symbol must be defined at link time either by linking the object file containing the imported symbol with another object file defines the symbol, or by placing a definition of the symbol in a linker command file using the let or set commands. The linker will display an error message if no object file or linker command file contains a definition of the imported symbol.

The second type of imported symbol has a non−zero value in the **sy_value** field. In this case, if no other object file defined the symbol at link time, the linker will define the symbol and allocate space to it (the symbol is assumed to be a variable). The amount of space allocated to the symbol is given in the

sy_value field. The symbol is allocated in the first linker output section that has the **comms** modifier as defined in a linker command file. If multiple imported symbols of the same name with non−zero values are defined in several object files then the linker will allocate enough space to hold the largest one encountered.

Mnemonic	Section Number	Meaning
N_ABS	−1	Absolute Symbol
N_UNDEF	0	Undefined external symbol
N_SCNUM	1 − 32767	Section number where symbol was defined

Object symbols having certain classes are also restricted to certain section numbers. These restrictions are summarized in the following table:

Class	Section Number
IC_EXT	N_ABS, N_UNDEF, N_SCNUM
IC_STAT	N_ABS, N_SCNUM
IC_LABEL IC_FENTRY IC_FEND IC_FEXIT IC_BBEGIN IC_BEND	N_SCNUM

Global and static symbols

Global and static symbol table entries are used to describe objects at physical memory addresses.

The format for a global or static symbol table entry is shown below:

Bytes	Type	Name	Description
0−3	long int	sy_namendx	string index of name
4−7	long int	sy_address	relocatable address
8−11	long int	sy_typendx	symbol index of symbol's type
12−15	long int	sy_scnum	N_UNDEF (externals only) N_ABS, N_SCNUM
16	char	sy_class	IC_EXT or IC_STAT

17	char	sy_flags	SF_FUNC

sy_namendx represents the string index of the symbol name in the string table. **sy_namendx** is set to zero for anonymous names. The **sy_address** field contains the physical address of the object.

The type of the object is described by the symbol table entry pointed to by **sy_typendx**.

sy_scnum is set to **N_ABS** if the symbol is absolute or **N_SCNUM** if the symbol is defined in a section and resolved. If the symbol has a storage class of **IC_EXT**, **sy_scnum** can also be set to **N_UNDEF** if the symbol is undefined.

The only information used in **sy_flags** is **SF_FUNC**. This flag indicates that the symbol was defined with an **fbegin** assembler directive. Source level debugging programs make use of this information.

Label symbols

Label symbol table entries are used to describe labels defined in Introl−C.

The format for a label type symbol table entry is shown below:

Bytes	Type	Name	Description
0−3	long int	sy_namendx	string index of name
4−7	long int	sy_address	relocatable address
12−15	long int	sy_scnum	N_SCNUM
16	char	sy_class	IC_LABEL

sy_namendx represents the string index of the symbol name in the string table.

sy_address represents the physical memory address of the label.

The value of **sy_scnum** is always set to **N_SCNUM**, in which represents the section n which the label was defined.

Function and block boundary symbols

Function and block boundary symbols table entries are used to describe the physical memory address ranges of high−level language functions and lexical blocks for debugging requirements. This information is provided automatically by Introl−C.

The format for a function and block boundary symbol table entry is shown in the following table:

Bytes	Type	Name	Description
0−3	long int	sy_namendx	string index of name
4−7	long in	sy_address	relocatable address

12−15	long int	sy_scnum	N_SCNUM
16	char	sy_class	IC_FENTRY, IC_FEND IC_FEXIT, IC_BBEGIN, IC_BEND

sy_namendx represents the string index of the symbol name in the string table. **sy_namendx** is set to zero for anonymous names. Normally function boundary symbols (those with the storage classes **IC_FENTRY**, **IC_FEND**, and **IC_FEXIT** will be anonymous). The symbol name for **IC_BBEGIN** symbols will be anonymous or the name of the block the symbol represents, while **IC_BEND** class symbols will always be anonymous.

The contents of **sy_address** will represent an in memory address of the given debugging point.

sy_scnum will represent the section under which the debugging symbol was defined.

Global or static symbols that mark the physical beginning of a function or procedure will have the **SF_FUNC** flag set in **sy_flags**. This symbol is the name of the function. For each function, a special symbol is put between the function name and the first local symbol of the function in the symbol table. This symbol has the storage class of **IC_FENTRY** and represents the entry point of the function after any language or target specific initialization code. Also, another special symbol is put immediately after the last local symbol of that function in the symbol table. This symbol has the storage class of **IC_FEND** and represents the address of the next byte in memory after the function.

Debugging symbols with a storage class of **IC_FEXIT** represent addresses of exit points in a function. Multiple **IC_FEXIT** points may be present in a function.

Blocks inside a function are marked with debugging symbols with a storage class of **IC_BBEGIN** for block begin and **IC_BEND** for block end. Local symbols defined inside of blocks will be placed between matching **IC_BBEGIN** and **IC_BEND** symbols in the symbol table.

The sequence for debugging symbols is shown below:

Symbol Name	Storage Class
function name	IC_EXT or IC_STAT
−	IC_FENTRY
local symbols	
−	IC_BBEGIN
symbols local to this block	
−	IC_BEND
−	IC_FEND

Type Symbols

Type symbols are used to describe the type, in high−level language terms, of an object described by an object symbol. Predefined constants called *atomic types* are used in fields of type symbols to directly describe the in−memory representation of data. The table following represents the list of predefined atomic machine types.

Name	Value	Description	size in bytes
IVOID	0	void type	0
ISCHAR	1	signed character type	1
IUCHAR	2	unsigned character type	1
ISI8	3	signed eight bit integer	1
ISI16	4	signed 16 bit integer	2
ISI32	5	signed 32 bit integer	4
IUI8	6	unsigned eight bit integer	1
IUI16	7	unsigned 16 bit integer	2
IUI32	8	unsigned 32 bit integer	4
ISFLOAT	9	single precision IEEE floating point	4
IDFLOAT	10	double precision IEEE floating point	8
ISI64	11	signed 64 bit integer	8
IUI64	12	unsigned 64 bit integer	8

The following sections describe the formats of symbol table entries used for type symbols.

Basic data types

Basic data types are used to define target specific data types which represent atomic data types in Introl−C (such as int). The symbol table entry for a basic type is shown below:

Bytes	Type	Name	Description
0−3	long int	sy_namendx	string index of name
16	char	sy_class	IC_SIMPLE
17	char	sy_flags	SF_BASIC

sy_namendx represents the string index of the symbol name in the string table.

The value of the **SF_BASIC** field of **sy_flags** defines the in memory representation of the data type.

Subrange types

Subranges are a type that consists of a range of legal values that are a subset of some other type.

The format of a subrange symbol table entry is shown following:

Bytes	Type	Name	Description

0−3	long int	sy_namendx	string index of name
4−7	long int	sy_from	low value of subrange
8−11	long int	sy_typendx	symbol index of underlying type
12−15	long int	sy_to	high value of subrange
16	char	sy_class	IC_SUBR
17	char	sy_flags	SF_BASIC

sy_namendx represents the string index of the symbol name in the string table. **sy_namendx** is set to zero for anonymous names.

The **sy_from** and **sy_to** fields contain the low and high values, respectively, that can be contained in the subrange. If the subrange is of an enumerated type then the low and high values are the ordinal values of the enumerated literals.

The **sy_typendx** field contains a symbol pointer to the type on which the subrange is based.

The value of the **SF_BASIC** field of **sy_flags** defines the in memory representation of the subrange data type. Note that a subrange of a given type may have a different (smaller) memory representation than the memory representation of the type on which the subrange is based. **sy_class** is always set to **IC_SUBR**.

Enumeration types

Enumeration type symbol table entries are used to describe enumerations defined in Introl−C. Enumeration types are described using a linked list of symbol table entries.

The first symbol table entry in the linked list has a storage class of **IC_ENUM** and the next symbol in the linked list is pointed to by **sy_firstndx**. The rest of the symbols in the linked list have storage classes of **IC_LITERAL**.

The format of an enumeration symbol table entry is shown below.

Bytes	Type	Name	Description
0−3	long int	sy_namendx	string index of name
12−15	long int	sy_firstndx	symbol index of first literal
16	char	sy_class	IC_ENUM
17	char	sy_flags	SF_BASIC

sy_namendx represents the string index of the symbol name in the string table. **sy_namendx** is set to zero for anonymous enumerations.

sy_firstndx points to a symbol table entry with a storage class of **IC_LITERAL**. Each literal points to the next literal using the symbol index contained in **sy_nextndx**. The list continues with a series of literals until **sy_nextndx** is zero.

The value of the **SF_BASIC** field of **sy_flags** defines the in memory representation of the enumerated data

type.

C array types

The format for a symbol table entry describing *C array types* defined in Introl−C is shown in below.

Bytes	Type	Name	Description
0−3	long int	sy_namendx	string index of name
4−7	long int	sy_size	size in bytes
8−11	long int	sy_elementndx	symbol index of element type
12−15	long int	sy_dimen	number of elements
16	char	sy_class	IC_CARRAY

sy_namendx represents the string index of the symbol name in the string table. **sy_namendx** is set to zero for anonymous C arrays.

The contents of **sy_dimen** represent the number of elements in the array.

The type of each element of the array is described by the symbol pointed to by **sy_elementndx**.

The total size of the array in bytes is kept in **sy_size**.

Pointer types

Pointer type symbol table entries are used to describe pointer types defined in Introl−C.

The format for a pointer type symbol table entry is shown here.

Bytes	Type	Name	Description
0−3	long int	sy_namendx	string index of name
8−11	long int	sy_typendx	symbol index of types
16	char	sy_class	IC_POINTER
17	char	sy_flags	SF_BASIC

sy_namendx represents the string index of the symbol name in the string table. **sy_namendx** is set to zero for anonymous pointer types.

The value of **sy_typendx** represents a pointer to the actual type pointed to.

The value of the **SF_BASIC** field of **sy_flags** defines the in memory representation of the pointer.

Tag types

Tag type symbol table entries are used to describe structure and union tags defined in Introl−C.

The format for a tag type symbol table entry is shown.

Bytes	Type	Name	Description
0−3	long int	sy_namendx	string index of name
8−11	long int	sy_typendx	symbol index of type the tag represents
16	char	sy_class	IC_TAG

sy_namendx represents the string index of the symbol name in the string table.

The value of **sy_typendx** represents a pointer to the type represented by the tag.

Set types

Set type symbol table entries are used to describe set variables.

The format of a set type symbol table entry is:

Bytes	Type	Name	Description
0−3	long int	sy_namendx	string index of name
8−11	long int	sy_typendx	symbol index of type of member's subrange
16	char	sy_class	IC_SETOF
17	char	sy_flags	SF_BASIC

sy_namendx represents the string index of the symbol name in the string table. **sy_namendx** is set to zero for anonymous set types.

The **sy_typendx** field represents a symbol index of a type which describes type of the set.

The value of the **SF_BASIC** field of **sy_flags** defines the in memory representation of the set.

Type−of types

Type−of symbol table entries are used to make new named types. The format of a type of symbol table entry is shown below.

Bytes	Type	Name	Description
0−3	long int	sy_namendx	string index of name
8−11	long int	sy_typendx	symbol index of type of member's subrange

16	char	sy_class	IC_TYPE
17	char	sy_flags	SF_BASIC

sy_namendx represents the string index of the symbol name in the string table.

The **sy_typendx** field is the symbol index of the type of the symbol.

Function and procedure types

Function and procedure type symbols describe the *return type* (if any) and types of all *parameters* to a function. A function symbol will have a storage class of **IC_EXT** or **IC_STAT** and use the **sy_typendx** field to point to a function type symbol table entry.

Function parameters are kept in a zero terminated linked list pointed to by **sy_firstndx**.

The format for a function symbol table entry is:

Bytes	Type	Name	Description
0−3	long int	sy_namendx	string index of name
8−11	long int	sy_typendx	symbol index of return type
12−15	long int	sy_firstndx	symbol index of parameter type list
16	char	sy_class	IC_FUNC

sy_namendx represents the string index of the symbol name in the string table. **sy_namendx** is set to zero for anonymous function types.

The function or procedure type symbol table entry uses **sy_typendx** to point to the type of the return value. Functions that do not return any value will have a zero index in **sy_typendx**.

Functions without parameters will have **sy_nextndx** set to zero, otherwise it will point to the first parameter type in a linked list of parameter types.

Structure, record, and union types

Structure, record, and union type symbol table entries are used to describe structures, unions, and records.

Members for structures, records, and unions are kept in a zero terminated linked list containing both member and bit field special symbols. This linked list is pointed to by **sy_firstndx**.

The format for a structured type symbol table entry is shown below.

Bytes	Type	Name	Description
0−3	long int	sy_namendx	string index of name
4−7	long int	sy_size	size in bytes

12−15	long int	sy_firstndx	symbol index of first member
16	char	sy_class	IC_RECORD or IC_UNION

sy_namendx represents the string index of the symbol name in the string table. **sy_namendx** is set to zero for anonymous structure, union, or record types.

All structured types that have members point to a linked list of members or bit fields using the **sy_firstndx** field, otherwise the **sy_firstndx** symbol index is zero. Each member in a structured type will have a byte offset representing how far the member is from the beginning of the structure, record, or union. Unions defined in Introl−C will have members that have byte offsets of zero.

sy_size represents the size in bytes of the whole structure type.

Special Symbols

Special symbols are used to describe offsets, register numbers, constants, bit fields or source file names.

Section symbols

A section symbol describes the size and address of the ICOFF sections created in each object file.

The format for a section symbol table entry is shown below:

Bytes	Type	Name	Description
0−3	long int	sy_namendx	string index of name
4−7	long int	sy_address	relocatable address
8−11	long int	sy_secsize	the size of the section
12−15	long int	sy_scnum	the output section number
16	char	sy_class	IC_SEC
17	char	sy_flags	

sy_namendx represents the string index of the symbol name in the string table. **sy_namendx** is set to zero for anonymous names. The **sy_address** field contains the physical address of the object.

The type of the object is described by the symbol table entry pointed to by **sy_typendx**.

The size of the section is contained in **sy_secsize**.

sy_scnum is set to the number of the output section that contains the section if the file has been linked.

The **sy_flags** is unused.

Local symbols

Local symbol table entries are used to describe the position of local variables either by an offset from a stack or frame pointer, or by a register number.

Local symbols are always grouped in the symbol table inside of the function in which they are defined.

The format for an local symbol table entry is shown below.

Bytes	Type	Name	Description
0−3	long int	sy_namendx	string index of name
4−7	long int	sy_offset	symbol's offset or register number
8−11	long int	sy_typendx	symbol index of symbol's type
16	char	sy_class	IC_LOCAL
17	char	sy_flags	SF_REG

sy_namendx represents the string index of the local symbol name in the string table.

The value of **sy_offset** represents a machine dependent offset for a local variable, either from a current stack pointer or frame pointer value. If the **SF_REG** flag is set in **sy_flags**, **sy_value** will contain a machine dependent register number.

The type of the object is described by the symbol table entry pointed to by **sy_typendx**.

Parameters

Parameter symbol table entries are used in conjunction with a function type symbol table entry, and are always in a linked list starting with a function type symbol.

The format for a parameter symbol table entry is shown below.

Bytes	Type	Name	Description
0−3	long int	sy_namendx	string index of name
4−7	long int	sy_offset	symbol's offset or register number
8−11	long int	sy_typendx	parameter type
12−15	long int	sy_nextndx	symbol index of next parameter
16	char	sy_class	IC_PARM
17	char	sy_flags	SF_REG, SF_ALIAS

sy_namendx represents the string index of the parameter name in the string table. **sy_namendx** is set to zero for anonymous parameters.

The value of **sy_offset** represents a machine dependent offset for a parameter variable, either from a current stack pointer or frame pointer value. If the **SF_REG** flag is set in **sy_flags**, **sy_value** will contain a machine dependent register number.

The parameter symbol list for a function is maintained through **sy_nextndx**, using a zero symbol index to mark the end of the list.

The type of the parameter is pointed to by **sy_typendx**. If the **SF_ALIAS** flag is set in **sy_flags**, **sy_typendx** will represent a symbol index of the non−local symbol referenced by this parameter.

The contents of **sy_flags** are used to determine if the parameter is kept in a register (**SF_REG**), or an alias for a non−local variable (**SF_ALIAS**).

Function prototypes defined in Introl−C using the ellipsis (...) for the last parameter are represented by having a zero symbol index for both **sy_typendx** and **sy_nextndx**.

Members

Member symbol table entries are used to describe the members of a structure, union, or record type.

Member symbols are kept in a zero terminated linked list headed by a symbol with a storage class of **IC_RECORD** or **IC_UNION**. Member types may be mixed with *bit field* symbols in the linked list.

The format for a member symbol table entry is:

Bytes	Type	Name	Description
0−3	long int	sy_namendx	string index of name
4−7	long int	sy_offset	offset in record
8−11	long int	sy_typendx	symbol index of type of member
12−15	long int	sy_nextndx	symbol index of next member
16	char	sy_class	IC_MEMBER

sy_namendx represents the string index of the symbol name in the string table.

sy_offset represents the byte offset in the containing structure of this member.

sy_typendx points to the type of this member.

Member symbols are kept in a linked list using **sy_nextndx** as a pointer to the next member symbol, with the value zero terminating the list.

Bit fields

Bit field symbol table entries are used to describe bit fields defined in Introl−C.

Bit field symbols are kept in a zero terminated linked list headed by a symbol with a storage class of **IC_RECORD** or **IC_UNION**. Bit field types may be mixed with member symbols in the linked list.

The format for a bit field symbol table entry is:

Bytes	Type	Name	Description
0−3	long int	sy_namendx	string index of name

4−7	long int	sy_offset	offset in record
8−11	long int	sy_typendx	symbol index of type of member
12−15	long int	sy_nextndx	symbol index of next member
16	char	sy_class	IC_MEMBER

sy_namendx represents the string index of the symbol name in the string table.

sy_offset represents the number of bits after the beginning of the structure that the bit field starts. The bit offset is contained in the lower 24 bits of **sy_offset**. The upper eight bits of **sy_offset** contain the size in bits of the bit field, from zero to 255 bits.

The field called **sy_typendx** points to the type of the bit field.

sy_nextndx is used to maintain the linked list of bit field or member symbols, with zero representing the end of the list.

Literal symbols

*Literal*s are symbol table entries used to describe enumeration literals.

The format of a literal symbol table entry is:

Byte	Type	Name	Description
0−3	long int	sy_namendx	string index of name
4−7	long int	sy_value	actual value
8−11	long int	sy_typendx	symbol index of type
12−15	long int	sy_nextndx	symbol index of next literal
16	char	sy_class	IC_LITERAL

sy_namendx represents the string index of the enumeration name in the string table.

sy_value represents the actual value of the enumeration literal.

The contents of **sy_typendx** represent a symbol index of the enumeration type that contains this symbol.

sy_nextndx points to the next literal in a zero−terminated linked list.

File symbols

File symbol table entries are used to describe names of source files and to mark the beginning of the symbols defined in the given source file in the symbol table.

File symbols can be found anywhere in the symbol table but precede the symbols in the object file that were defined in the given source file.

The format for a file type symbol table entry is shown below.

Bytes	Type	Name	Description
0−3	long int	sy_namendx	string index of name
4−7	long int	sy_timdat	last modified time of file
16	char	sy_class	IC_FILE

sy_namendx represents the string index of the symbol name in the string table.

String Table

Each symbol table entry has a string table index kept in the field called **sy_name**. This field will either contain a positive integer representing the string index of the symbol name or a zero representing an anonymous symbol. All symbol table names are stored contiguously in the string table with each symbol name terminated by a **nul** (zero byte). The first byte of the string table is a **nul**. Offsets into the string table are greater than or equal to one.

For example, given a file containing two symbols *main* and *counter*, the string table has the format as shown below:

0	`m'	`a'	`i'
`n'	0	`c'	`o'
`u'	`n'	`t'	`e'
`r'	0		

The string index of *main* is 1 and the string index of *counter* is 6.

Motorola S records

S record files are printable files, consisting any number of the eight different record types. S record files can be created from an ICOFF input file by **ihex**. Each record has the following structure:

Field	Size in characters
Type	2
Record Length	2
Address	4, 6, or 8
Code / Data	0 − N
Checksum	2

The record length, address, code/data, and checksum fields are hexadecimal bytes coded in ASCII: 00 is the value 0, 01 the value 1, etc. The record length is the number of data bytes (two hex digits) including the address, code/data, and checksum fields. The checksum field contains the one's complement of the sum of all

the bytes from the record length field through the end of the code/data field.

Each record can be terminated with either a carriage return, line feed, or a **nul** (zero) character.

An example assembly file:

```
org     $1000
dc.b    'some data',0
org     $10000
dc.b    'data at a higher address',0
```

run through ihex produces:

```
S10D1000736F6D6520646174610074
S21D0100006461746120617420612068696768657220616464726573730034
S9030000FC
```

The record types are as follows:

S0
: Header record for each block of S records which may contain descriptive information identifying the following block of S records. The address field is typically zeros. The code/data field of an S0 record typically contains a "hexified" ASCII string.

S1
: Code or data record containing a 2−byte address (4 hex digit address).

S2
: Code or data record containing a 3−byte address (6 hex digit address).

S3
: Code or data record containing a 4−byte address (8 hex digit address).

S5
: Count record containing the number of S1, S2, and S3 records transmitted in a block. The count appears in the address field, and there is no code/data field (I never saw this actually used).

S7
: Termination record for a block of S records. A 4−byte address contains the address where execution starts. There is no code/data field.

S8
: Termination record for a block of S records. A 3−byte address contains the address where execution starts. There is no code/data field.

S9
: Termination record for a block of S records. A 2−byte address contains the address where execution starts. There is no code or data field.

Intel hex

Intel hex record files are printable files consisting of any number of Intel hex records. Intel hex files can be created from ICOFF files by **ihex**. Each record has the following format:

```
:CCAAAARR...ZZ
```

where:

CC
: Number of data bytes

AAAA
: Address field

RR
: Record type:
  - ◊ 00 data record
  - ◊ 01 end record
  - ◊ 02 paragraph record
  - ◊ 03 transfer address record

...
: Data field

ZZ
: Checksum for data record

All fields in an Intel hex record are in hexadecimal with two hex digits representing a byte. The **CC** field is the actual number of bytes in the data field (the address, record type, and checksum bytes are not included in the count). The checksum is the two's complement of the sum of the bytes from the number of data bytes field through the last byte of the data field.

An example assembly file:

```
org     $1000
dc.b    'some data',0
org     $10000
dc.b    'data at a higher address',0
```

run through ihex (with the **−gi** option) produces:

```
:0A100000736F6D6520646174610078
:020000021000EC
:190000006461746120617420612068696768657220616464726573730003A
:04000003000000000F9
:00000001FF
```

Tektronix hex

Tektronix hex record files are printable files consisting of any number of Tektronix hex records. There are three types of records: data records, termination records, and abort records. ihex with the **−gt** option converts **ICOFF** files to Tektronix hex.

Data records have the following format:

Field	Size
/	1
Address field	4
Byte count	2
First checksum	2
Data field	0 − N
Second checksum	2

Termination records have the following format:

Field	Size
/	1
Address field	4
00	2
Checksum	2

Abort records have the following format:

Field	Size
/	1
/	1
Message	1−69

Extended Tektronix hex

Extended Tektronix hex record files are printable files consisting of any number of extended Tektronix hex records. ICOFF files can be converted to extended Tektronix hex files with **ihex**.

Each block of records begins with a six−character header record. A block can be up to 255 characters long, not counting the end−of−line. A header field has the following format:

Field	Size	Description
%	1	Specifies Extended Tekhex format
Block length	2	Number of characters in the block
Block type	1	6 = data block 3 = symbol block 8 = termination block
Checksum	2	Checksum for the record

A data record has the following format:

Field	Size	Description
Header	6	Standard header field
Address field	2 − 17	Address where the object code is to be loaded
Data field	0 − N	Data

A termination record has the following format:

Field	Size	Description
Header	6	Standard header field
Transfer address	2 − 17	Address where program execution should start

A symbol record has the following format:

Field	Size	Description
Header	6	Standard header field
Section name	2 − 17	Name of the section containing the symbols in this block
Section definition	5 − 35	At least one of these must appear for each section. See the section definition field description below
Symbol	5 − 35	Zero or more symbol definitions as described below

definitions	each	

A section definition field has the following format:

Field	Size	Description
0	1	A zero specifies a section definition field
Address field	2 − 17	Starting address of the section
Length	2 − 17	Length of the section

A symbol definition field has the following format:

Field	Size	Description
Type	1	1 = global address 2 = global scalar 3 = global code address 4 = global data address 5 = local address 6 = local scalar 7 = local code address 8 = local data address
Symbol	2 − 17	Variable length symbol
Value	2 − 17	Value associated with the symbol

The IDB Debugger

idb is the old CODE debugger, having been replaced by the GUI Debugger. We supply **idb** and its documentation for the people who have used **idb** in the past. **idb** is no longer actively supported and may be dropped from future releases of CODE. **idb** has a configuration file named **idbrc**, which is responsible for decoding command line options and setting internal variables.

Command line
: How to run **idb**.

Command syntax
: How to type commands at the command prompt.

Idb commands
: Commands that can be entered at the command prompt or in a debug procedure.

Configuration files and variables
: The idb configuration files and the variables defined and used by **idb**.

Configuring idb targets
: How to configure idb for the targets it supports.

Command line

```
idb [options] filename
```

−g filename

Read in *filename* instead of the default configuration file *idbrc*.

−k

Print **idb**'s version number. If **idb** is invoked with no arguments, it will print the version number of its drivers.

−X

Any option other than those above that are specified on the command line set an **idb** variable called **$arg*X*** where the *X* is replaced by the command line option. For example, specifying **−m1** on the command line would set the variable **$argm** equal to 1.

idbrc related options

The following options are interpreted by code contained in *idbrc*:

−c

When specified, the coprocessor's registers will also be displayed by the registers command.

−f

Define several function keys by reading the *idbfkeys* file, and display some of the definitions on the bottom line of the screen.

−i

Trap input and output to *stdin* and *stdout* and redirect it to the debugger command window. This option requires that you are using the standard low level input output routines as described in the runtime environment.

−m number

Set target *processor type*. The supported processors are:

Number	Processor
01	6801
03	6301
05	68HC05
08	68HC08
09	6809
10	68010
11	68HC11
12	68HC12
16	68HC16
20	68020
30	68030
68	683XX
68000	68000

−p

Do not go into *pass through* mode when the debugger is invoked. Pass through mode connects the host directly to the target environment for those target environments that allow direct command input.

−r

Do a **load** and **rgo** when idb starts. This option also implies the **−i** option. The −r option allows you to run a program immediately:

```
idb −r −m11 program
```

will run the program immediately with the simulator, print simulated output and send keyboard input to the simulated program.

−t number

Set *target type*. If a target is not specified, the default is the software simulator. The supported targets are:

Number	Target Description
1	Introl Monitor Interface (IMI)
2	68HC11 (6801, 6301) Simulator
3	6809 Simulator
4	68HC16 Simulator
5	68XXX Simulator
6	68HC12 Simulator
7	68HC05 Simulator
8	68HC08 Simulator
16	68HC16 Background Debug Mode
200	HMI 200 Series Emulator
333	683XX Background Debug Mode
600	Pentica MIM−600 Emulator

Note that some of the targets may not be supported on all host systems. The 68HC16 and 683XX Background Debug Modes may not be available on host systems that do not have a parallel port or where the parallel port is not directly accessable by **idb**. **idb** run with no options will display a list of targets that are supported on a given host.

−v number

Set the video mode for MS−DOS hosts. If *number* is 1, the display is set to 80 columns and 60 rows; if *number* is 2, the display is set to 132 columns and 25 rows.

−w [number]

Without a numeric argument will start the debugger in windowing mode with the default window layout. A numeric argument for **−w** allows any predefined window layout to be chosen. The valid window layout numbers are:

Number	Style
1	Normal Window Set (default)
2	Source window contains mixed source and disassembled code.

3	Source window contains disassembled code.
4	Two source windows, one containing source and the other containing disassembled code.

Command syntax

The command prompt is used to indicate that **idb** is ready to access input and interpret commands. Case is significant for command entry and all predefined commands are lower case. Commands are generally entered by typing a command followed by zero or more comma separated arguments. Commands may be abbreviated to the shortest unique string. If an entered string is ambiguous the debugger will display a list of the commands that could match the string.

idb supports a comment character (#) that causes anything following it on the input line to be ignored. A comment character occurring inside of an identifier or a string looses its special meaning.

To interrupt long running commands, press control−C or, on MS−DOS hosts, control−break. When the interrupt character is pressed, execution of the target program is interrupted wherever it happens to be executing. The source and register windows and status line will be updated to reflect the new processor state.

The following paragraphs introduce concepts used in the command descriptions and their arguments. The word in ***bold italic*** is used in the meta−notation in those descriptions.

An ***integer*** in idb can be represented in decimal, octal, or hexadecimal. A number with the digits zero through nine (0 − 9) that does not have a leading zero is interpreted as decimal. A number starting with zero and containing digits zero through seven (0 − 7) in it is interpreted as octal. A number starting with a ``0x" or ``0X" is interpreted as a hexadecimal constant. Hexadecimal constants consist of the digits zero through nine, and the letters ``A" through ``F" in upper or lower case.

A ***float*** can be represented as a decimal whole part, followed by a decimal point, followed by a fractional part, followed by the letter ``e" or ``E", followed by a signed power of ten exponent. Either the decimal point and the fractional part, or the ``e" and the exponent can be left out of a floating−point constant.

An ***identifier*** is a sequence of characters starting with an alphabetic character or an underscore followed by any number of alphabetic characters, numeric digits, and the underscore character. Any characters enclosed in single quotes is also an identifier.

C identifiers can be used in expressions if the value is known in the current context. Identifiers which are names of functions have a value of the address of the function's entry point after any initialization code.

idb variables are identifiers whose names have a leading dollar sign (**$**). They have integer values which are local within the debugger, can be used for any purpose, and can be used in expressions along with C identifiers. There are some **idb** variables used within **idb** to control its operation, see Variables.

Identifiers preceded by a percent sign (%) are understood to be ***register names***. Register names can be used in expressions and be read or written.

A ***string*** is a sequence of characters enclosed in double quotes. If an identifier is used in place of a string, the debugger interprets the identifier as a string.

An ***expression*** can be arbitrarily complex and involve program variables, **idb** variables, processor registers,

and constants.

Binary and Unary Operators

Expressions are made up of program variables, debugger variables, and processor registers that are operated on with operators. Most operators in **idb** are binary or unary. All binary operators take two expressions as operands. An expression given as an argument to a binary operator can usually evaluate to either an integer or real value, although some operators work only on integer operands. The binary operators are shown in the following table.

Operator	Description
=	Assignment
:=	Assignment
||	Logical OR
&&	Logical AND
|	Bitwise OR
&	Bitwise AND
^	Bitwise exclusive OR
==	Test for equality
!=	Test for inequality
<	Test for less than
>	Test for greater than
<=	Test for less than or equal to
>=	Test for greater than or equal to
<<	Left shift
>>	Right shift
+	Addition
−	Subtraction
*	Multiplication
/	Division
%	Modulus

Assignment operators can be used to assign values to program variables, debugger variables, or processor registers. Logical and comparison operators evaluate to a value of one for true and zero for false. The bitwise operators, the shift operators, and the modulus operator work on integer arguments only.

The unary operators are as follows:

Operator	Description

!	Logical negation
~	One's complement
+	Unary plus
−	Unary minus
*	Pointer dereference
&	Address of

The one's complement operator works on integers only. The pointer dereference operator works on an expression that evaluates to a pointer type.

The ``address of'' operator can be used on any symbol or expression that evaluates to an addressable object in memory. This operator returns the address of the addressable object. The type of the result is a pointer to the object in memory.

Miscellaneous Operators

idb supports an additional set of operators that are not quite binary nor unary:

Operator	Description
address[*integer*]	Array reference
@number	Address of source line *number*
string@number	Address of source line *number* in file *string*
string.identifier	Qualified identifier
address.identifier	Structure member reference
address−>identifier	Structure member reference

The array reference operator is used to index into an array. The address should evaluate to an array or a pointer. If address evaluates to a pointer, then the value of the pointer is used as the base address of the array. The integer is evaluated and multiplied by the size of each array element or item that the pointer refers to. This value is then added to the base address to give the final address to reference.

The address of source line operator can be used to get the address of a source line in an expression. If the operator is used as a unary operator, *number* is assumed to be a source line number in the current source file. If the operator is used as a binary operator, then *string* should be the name of a source file. *number* is then assumed to be a source line in the named file. It is an error to use the address of source line operator on a source line that has no executable code associated with it.

The qualified identifier operator is used to specify an identifier more directly than the default scoping rules. *string* can be the name of a source file or a function. The identifier is then searched for in the context represented by *string*.

The structure member reference operators are used to access members of a structure whose address is given by *address*. The address must be an expression that evaluates to a structure, or to a pointer to a structure. If *address* evaluates to a pointer, then the pointer's value is used as the base address of the structure.

Built−in functions

idb has several function−like constructs that can be used in expressions and are described below.

The names of the debugger functions that have vertical bars in them represent a family of functions. For example, the **$get** function can be used as **$getb** or $**getub**.

$defined(identifier)
: Returns a non−zero value if the *identifier* is an extern or static defined symbol in the object file.

$endof(identifier)
: The address of the end of the *identifier* section is returned. Zero is returned if the section does not exist.

$getchar()
: Read a character from the keyboard and return its ASCII value.

$putchar(integer)
: Write a character to the current output window.

$get[b|ub|c|uc|w|uw|l|ul|f|d] (address)
: Read a byte, unsigned byte, character, unsigned character, word, unsigned word, long, unsigned long, float or double from the specified *address* in the target environment.

$put[b|ub|c|uc|w|uw|l|ul|f|d] (address,expression)
: Write a byte, unsigned byte, character, unsigned character, word, unsigned word, long, unsigned long, float or double *expression* to the specified *address* in the target environment.

$realvalue(identifier)
: The value of the external, or static symbol identifier is returned. If identifier represents a function, the address of the function's first instruction is returned.

$startof(identifier)
: The address of the beginning of the section *identifier* is returned. Zero is returned if the section does not exist.

$sizeof(identifier)
: The size of the section *identifier* is returned. Zero is returned if the section is empty.

$validprocessor(integer)
: Returns a non−zero value if the given *integer* matches the identifier of a supported microprocessor.

$validtarget(integer)
: Returns a non−zero value if the given *integer* matches the identifier of a supported target environment.

$windowlastcurrent(identifier)
: Returns the current line number associated with the last source display in the window named *identifier*.

$windowlastfile(identifier)
: Returns the file number associated with the last source display in the window named *identifier*.

$windowlastlow(identifier)
: Returns the line number of the first source line associated with the last source display in the window named *identifier*.

$windowlasthigh(identifier)
: Returns the line number of the last source line associated with the last source display in the window named *identifier*.

Idb commands

There are several types of **idb** commands.

Control and Function keys

idb recognizes certain control and function keys. Control keys are recognized by all versions of **idb**. Function keys are recognized by the MS−DOS version only. You can also create your own control or function key procedures. **idb** will execute the following procedures when the corresponding control key is pressed. The procedures are undefined by default:

Name	Value
keyNUL	0x00
keyctrlA	0x01
...	...
keyctrlZ	0x1A
keyESC	0x1B
keyFS	0x1C
keyGS	0x1D
keyRS	0x1E
keyUS	0x1F
keyDEL	0x7F

idb will execute the following procedures when the corresponding function key is pressed. The procedures are undefined by default.

Procedure	Function Key
keyHOME	HOME
keyUP	up arrow
keyDOWN	down arrow
keyLEFT	left arrow
keyPPAGE	PAGE UP
keyNPAGE	PAGE DOWN

keyIC	INSERT
keyDC	DELETE
keyF0	F1
...	...
keyF63	F63

The following control key definitions are created in the *idbrc* file:

Key	Description
control−B	Set breakpoint at current line
control−D	Move current line down one line
control−N	Execute a next command
control−R	Redraw the screen
control−U	Move current line up one line

If the **−f** option is specified, the following function key definitions are created in the idbfkeys file:

Key	Description
F1	Execute a next command
F2	Execute a step command
F3	Execute an istep command

General

commands
Show a list of **idb** commands. The commands are grouped by function.

command identifier
List the specified user defined command procedure.

load
Download the object file to the target environment. Load can be executed any number of times during a debugger session.

pass [integer]
Enter *pass through* mode. Pass allows direct communication with the target environment, and works only with environments that have a command set that can be entered with a terminal. The *integer* argument is the escape character which defaults to control−D (ASCII 0x4). The escape character is interpreted as the ASCII value of the character used to terminate pass through mode.

quit
Leave **idb** and return to the host operating system.

source string
Read and execute an **idb** command file. Command files may contain any legal **idb** commands. The debugger will search the current working directory for a file with the name *string*. If no file by that name is found, the **lib** directory in the Introl directory hierarchy will be searched. Finally, the

config directory in the hierarchy will be searched. **idb** will display an error message if the file is not found in any of the standard directories.

system
Escape temporarily to the host operating system.

system string
Execute the command line in *string* under the host operating system.

Printing and expressions

expr expression
Evaluate *expression*. This command is necessary to perform an assignment. The period (.) is an alias for **expr** when used alone at the beginning of an input line.

fprint string[,expression]
Print *expression* with the given format. The characters in *string* are printed normally until a percent sign (%) is encountered. The percent sign is the lead−in character for a format specifier. The command is similar to the C language *printf* function with the following format modifiers:

Character	Description
c	Print an integer as a character
d	Print an integer in decimal
o	Print an integer in octal
x	Print an integer in hexadecimal
s	Print the bytes in the target system's memory space at an address as a string (null terminated)
e	Print a real number in exponential floating−point format
f	Print a real number in Fortran type floating−point format
g	Print a real number in the narrower of %e or %f

fregisters
Display floating point registers on target systems that have arithmetic coprocessors.

print [expression{,expression}]
Print an expression or expressions.

registers
Display the processor registers.

sections
Display program sections. All program section names, origins, and sizes in the current ICOFF file are displayed.

show
Display all currently defined debugger variables and their values.

whatis expression
Describe an address or symbol. **whatis** prints out **idb**'s idea of what an identifier or the type of an expression is.

where [integer]
Show current program location. If the target processor maintains a runtime linked list stack frame, then the dynamic function call history will be displayed up to the main entry point. The *integer* argument indicates how many stack frames should be traversed by the **where** command. By default, all active stack frames will be displayed.

Source file manipulation

back

Search source file backward for the last pattern. The question mark (**?**) is an alias for back when used alone at the beginning of an input line. **back** searches the current source file from the current line backward, wrapping around from the beginning of the file to the end and ending the search, if no matching string is found, when the search again reaches the current line.

back pattern

Search source file backward for the given *pattern*.

file

Display the name of the current source file.

file filename

Set the current source file name to *filename*. Any text file can be made the current source file using this command.

file integer

Set the current source file by file index *integer*. The index number for a file can be determined with the **files** command.

files

Display a list of files and their index numbers. The files displayed by the **files** command are those that comprise the set of files referenced by the object file being debugged.

forward

Search the current source file for the last pattern. The slash (/) is an alias for the **forward** command when used alone at the beginning of an input line. **forward** searches the current source file from the current line wrapping around from the end of the file to the beginning and ending the search, if no matching string is found, when the search again reaches the current line.

forward pattern

Search the current source file for *pattern*.

list

List the source file in the context of the last **list** command, or from the first line in the current source file if there was no last **list** command.

list identifier

If *identifier* is a C function, list the source code in that function. If there is no function of that name, list the source file named by *identifier.c*.

list integer[,integer]

List the current source file. The actual function of list varies depending on whether or not **idb** is running in windowing mode. In windowing mode, the list command will display enough source lines to fill the source window. In non−windowing mode, the list command will display the source lines from the line number represented by the first integer to the line number represented by the second integer. If the second integer is omitted, then the debugger variable **$listlines** controls how many lines should be displayed.

listsource integer,integer

A low−level source listing command that is used by the **list** command in non−windowing mode. It displays the current source file from the line number represented by the first *integer* to the line number represented by the second *integer*. This command is not intended for common use.

use

Display the list of directories that will be searched for source files. The directories in this list will be displayed in the order in which they will be searched.

use string

Prepend the path *string* to the list of directories searched for source files.

Processor control

delete all
: Delete all **stop** and **when** breakpoints.

delete number[,number]
: Delete selected **stop** and **when** breakpoints. The breakpoints specified or within the specified range will be deleted.

halt
: Stop the processor during a breakpoint sequence.

go
: Start the processor. The processor is run from the current context. If the processor has been reset, this command is equivalent to the **rgo** command.

go address
: Start the processor, stopping at the specified address. The processor is started at the current context, and will execute with an execution breakpoint set at the *address* specified. If the processor has been reset, then this command is equivalent to **rgo *address***. An identifier given as the *address* argument is interpreted by setting the breakpoint at the entry point of that function or procedure. An integer given as the *address* will be interpreted as an executable line number in the current source file. The program line number will be converted to the address of that line in the target processor's memory space. To refer to an explicit target memory address, you must use an expression like 0xE000+0 to reference an address rather than a line number.

istep
: Single step one machine instruction.

next
: Single step one source line, stepping over functions.

return [expression]
: Return from the current function. The optional *expression* will be returned by the function.

reset
: Reset the target processor.

rgo
: Reset the target processor and go.

rgo address
: Reset the target processor, set an execute breakpoint at *address*, and go.

step
: Single step one source line into functions.

stop
: Display a list of **stop** and **when** breakpoints and their index numbers.

stop at address
: Define an unconditional execution breakpoint. Execution will stop when this breakpoint is encountered. *address* is interpreted as in the **go** command.

stop at address if expression
: Define a conditional execution breakpoint. When the processor executes the instruction at the specified *address*, *expression* is evaluated. If *expression* evaluates to a non−zero value, then the stop event will occur, stopping the processor.

  The debugger variable **$count** is set to the iteration count of the current stop breakpoint and can be used in *expression*.

  The *address* argument will be interpreted as they are in the unconditional **stop** command.

stop [read|write|access] address

Define a data access breakpoint. An access breakpoint occurs when the address(es) specified with the *address* argument are read, written, or accessed (either read or written). This form of the stop command works only if the current target environment type supports access breakpoints.

If an access breakpoint is set and the address argument evaluates to a target address that has a C type associated with it, then the breakpoint will be set on every byte associated with an object of that type. For example, if address is an array expression, the breakpoint will occur if any element of the array is accessed.

stop [read|write|access] address if expression

Define a conditional data access breakpoint. This form of the stop command works only if the current target type supports access breakpoints.

The *address* argument is interpreted exactly as in the unconditional stop on access event command.

stop clock count

Stop the processor after *count* cycles have elapsed. This command only works if the target supports clock breakpoints.

stop clock count if expression

Stop the processor after count cycles have elapsed if expression is non−zero.

when

Display a list of **stop** and **when** breakpoints and their index numbers.

when at address
...
end

Define an unconditional execution command−point. If, during target program execution, the program attempts to execute the instruction at *address*, the **idb** commands defined as part of the **when** command will be executed. If the processor is to be halted, a **halt** command must be included in the commands executed.

Address is interpreted as in the **go** command.

when at address if expression
...
end

Define a conditional execution command−point. If, during the target program execution, the program attempts to execute the instruction at *address*, *expression* is evaluated. If *expression* evaluates to a non−zero value, then the **idb** commands defined as part of the **when** command will be executed.

when [read|write|access] address
...
end

Define a data access command−point event. The **idb** commands associated with the **when** command are executed when the address(es) specified with the *address* argument are read, written, or accessed (either read or write). This form of the **when** command works only if the current target environment type supports access breakpoints.

If an access breakpoint is set and the address argument evaluates to a target address that is a C type associated with it, then the breakpoint will be set on every byte associated with an object of that type.

Otherwise the *address* argument will be interpreted as they are in the unconditional **when** command.

when [read|write|access] address if expression

...

end

Define a conditional data access command−point. This form of the **when** command works only if the current target environment type supports access breakpoints.

The *address* argument is interpreted exactly as in the unconditional when on access event command.

when clock count

...

end

Define a clock command−point event. The **idb** commands associated with the **when** command are executed when *count* cycles have elapsed. This form of the **when** command works only if the current target environment type supports clock breakpoints.

when clock count if expression

...

end

Define a conditional clock command−point. This form of the **when** command works only if the current target environment type supports clock breakpoints.

Processor tracing and reverse execution

These commands are experimental in version 3.11 and may never be developed further in **idb**, the command line/screen oriented Introl debugger. The functionality will is available in the new CODE Debugger. Sorry about the odd interface to the tracing commands in **idb**. The **Execute** interface will be much more convienient and have greater functionality.

The instruction set simulators can maintain a *trace buffer* of the execution state of a program. The trace buffer, when enabled, records information about a programs execution. The following commands work in conjunction with the trace buffer:

trace on[,size]

Turn instruction tracing on. size specifies the number of instructions to keep in the trace buffer. The default size is 4096 instructions. The instruction count can be useful for determining how large a trace buffer is required. Be careful with the size argument: tracing can consume a lot of memory.

trace off

Turn instruction tracing off.

trace bgo[,target]

Execute a program backwards until the beginning of the trace buffer is reached, *target* is reached, or a breakpoint is encountered. This command is analogous to a reverse **go** command.

trace fgo[,target]

Execute a program forward untill the end of the trace buffer is reached, target is reached, or a breakpoint is encountered. This command is analogous to a **go** command except that the trace buffer is interpreted rather than re−executing instructions.

trace bi

Reverse execute the last machine instruction, like a reverse **istep**.

trace fi

Re−execute the next instruction, like **istep** except trace information is used.

trace bs

Reverse execute the last source line, like a reverse **step**.

trace fs

Re−execute the next source line, like **step** except trace information is used.

trace bn
Reverse execute the last source line, skipping over function calls like a reverse **next**.
trace fn
Re−execute the next source line, skipping over function calls, like **next** except trace information is used.
You can only reverse execute a program if tracing is turned on and some of the program has been executed. You can only re−execute instructions if those instructions have been reverse executed.

Scripting commands

break
Exit a **while** loop.
define identifier
...
end
A command procedure named *identifier* can be defined with the **define** command. All lines following the define command until an end command are saved as the body of the command procedure. A command procedure can be executed just like any other **idb** command. When a command procedure is redefined, the old definition is discarded. The list of command procedures is searched before the built−in command table so a command procedure definition can override that of a built−in command.

A comma separated list of arguments can be passed to a command procedure. Within the body of the procedure, the arguments can be referred to numerically with a dollar sign (?**$**?) followed by a decimal number indicating the argument number. Arguments are numbered from zero.

end
Used to indicate the end of a multi−line statement. It is used to terminate **when**, **define**, **if**, and **while** commands.
if expression
...
end
Evaluate *expression* and execute the following lines, until an **end** command is encountered, if *expression* evaluates to a non−zero value. Otherwise, the subsequent command lines until the **end** are ignored.
undefine identifier
Undefine a command procedure previously defined with the **define** command.
while expression
...
end
Evaluate the expression and, as long as the expression evaluates to a non−zero value, repeatedly execute the lines between the while command and the next−occurring end command.

Memory

disassemble [address{,number}]
Disassemble starting at *address*. If *number* is specified, **idb** will disassemble *number* machine instructions.
dump [address{,address}]
Dump a memory region in hexadecimal and ASCII.

fill address,address,integer
Deposit the byte value of *integer* into the memory region starting at the first *address* and ending at the second *address*.

move address,address,integer
Move the memory region starting at the first *address* to the second *address*. There will be *integer* bytes moved.

Configuration

passcommand string
Pass a configuration command string to the target driver.

port
Display the name of the current communication point and the communication baud rate (if applicable).

port string[,integer]
This form of the **port** command will open the communication port named *string* and set the baud rate to *integer*, if present. The communication port is dependent on the host system and target.

For serial port based targets on Unix, it should be a character special file associated with a serial port. Under MS−DOS it should be ``com1'' or ``com2''.

If string is ``parallel'', a parallel port at address *integer* is enabled.

processor
Display the current target processor type.

processor integer
Set target processor type to the processor specified by the integer argument. See the −m option for a list of valid processor types.

processors
Display a list of microprocessors supported by **idb**.

targettype
Display the currently active target environment type.

targettype integer
Set the current target type to integer. See the −t option for a list of valid target types.

targettypes
Display a list of the valid target environment types supported by this version of **idb**.

Windowing

A *boolean* can be one of **yes**, **on** or **true** for affirmation, or **no**, **off** or **false** for negation.

clear
Clear the current window

clear identifier
Clear the window named *identifier*.

displaysource
Display source in the current window around the source line number in **$currentline**. This is a low−level source display function which, in window mode, is normally called to maintain the source window.

displaysource integer

Display source in the current window around the source line number indicated by integer.

fillwindow identifier,integer

Fill the window named *identifier* with the character whose ASCII value is given by *integer*.

hilight boolean

The *boolean* argument of this command controls highlighting in the current window. If highlighting is turned on, then subsequent output to the current window will be in reverse video. Turning highlighting off makes subsequent output come out in normal video.

hilight boolean,identifier

The *boolean* argument of this command controls highlighting in the window named *identifier*.

moveto integer,integer

Move the cursor in the current window to the row indicated by the first *integer*, and the column indicated by the second *integer*. The upper left corner of a window is row zero, column zero.

redraw

Redraw the entire screen. **redraw** is useful if the screen contents become jumbled for some reason. The default definition of the control−R key, defined in key procedure **keyctrlR**, will perform a **redraw**.

setwinflags integer[,identifier]

Sets the type flag of the window to *integer*. A value of one will set the type to source code only, two will set the type to disassembled code only, and three is mixed source and disassembled code. If *identifier* is specified, that window's type flags are set, otherwise the current window will be used. This only applies to source windows.

scroll boolean

Control scrolling in a window. When scrolling in a window is turned on, a newline on the last line of the window causes the window to scroll up one line.

scroll boolean,identifier

Control scrolling in the window named *identifier*.

scrollpause boolean

Control whether a window will pause the output of a command when it is about to scroll off the top of a window. If enabled, a prompt will be displayed before the window is scrolled.

scrollpause boolean,identifier

Control the scroll pausing attribute of the window named *identifier*.

setwindow identifier

Set the window named identifier as the current window. The current window is the one from which command input is taken and to which command output is sent.

setvideomode integer

Set the BIOS video mode for MS−DOS hosts to *integer*.

startwindows

Initialize the **idb** windowing system. The **$LINES** and **$COLUMNS** debugger variables will be set to the size appropriate for the terminal after the windowing system as been initialized. **$LINES** will be set to the total number of lines on the display screen, and **$COLUMNS** will be set to the total number of columns. The **startwindows** command should be used no more than once in a debugging session.

tabstops identifier,integer

Set the tab stops of the window named *identifier* to be at every *integer* column. Tab stops are set every eight columns by default.

truncate boolean

Control whether output sent to the current window will be truncated at the right edge of the window. If truncation is turned off, then output that passes the right edge of a window will cause an automatic newline and the rest of the output will be displayed on the next window line.

truncate boolean,identifier

Control line truncation in the named window.

window identifier,identifier

Perform an operation on the window named in the first *identifier* argument. The operation performed is given in the second identifier argument. The various window operations may take one or more integer arguments in a comma separated list following the operation identifier. The following list enumerates the operations supported by the window command. In this list, the window operation and its arguments will be shown. Each window operation should be preceded by the window *identifier*, prefix specifying the window name with the identifier argument.

window identifier, new{[,integer,integer,integer]}

Create a new window. The *integer* arguments are, in order, the starting row, the starting column, the length of the window in rows, and the width of the window in columns. If the length and width arguments are omitted, then the length defaults to **$LINES**−row and the width defaults to **$COLUMNS**−column. This makes a window whose upper left hand corner is at (row, column), whose lower line is the bottom of the screen, and whose right column is the right−most column of the screen. If the row and column arguments are omitted, then (0, 0) is assumed. This is the upper left−hand corner of the screen. The last argument sets the window type. A value of one will set the type to source code only, two will set the type to disassembled code only, and three is mixed source and disassembled code.

window identifier delete

Delete the named window. If the named window is the current window, then all windows are deleted.

Configuration files and variables

idb reads the file *idbrc* automatically when it is invoked. *idbrc* initializes the debugger for operation. *idbrc*, in turn, may also read the file *idbport* to initialize the communication port and *idbfkeys* to initialize the definitions of function key procedures.

idbrc

The file *idbrc* is the general configuration file for **idb**. If *idbrc* is not found in the current directory then the $INTROL/config directory will be searched. When *idbrc* is found, **idb** will execute it.

Various command line options are interpreted by the commands in *idbrc* and are used to configure the debugger. The *idbrc* command file is mandatory: the debugger will not function if not configured. The following happens when *idbrc* is processed:

- The target microprocessor type and the target environment type are set.
- Various debugger variables are set as needed by the target environment.
- If the target environment needs a communication port, the port is opened. Pass through mode is entered if needed.
- Various general debugger variables are set.
- The default source listing command for non−windowing mode is defined.

idbport

The command file *idbport* is executed by the *idbrc* file to open the communication port. It contains commented port commands. Select the one appropriate for your host and target combination.

idbfkeys

The *idbfkeys* command file is read by the *idbrc* command file if the **−f** option is present on the idb command line.

Variables

These **idb** variables are used to control the operation of the debugger. Most of these variables are set to initial values in *idbrc*.

$clock, $clockhigh

These variables are set by targets that can do processor cycle counting. They represent the number of instruction cycles that the processor has executed. **$clock** is in the range 0..999,999,999 and **$clockhigh** is incremented each time **$clock** goes from 999,999,999 to 0.

The cycle counter is set to zero when the processor is reset.

$coprocessor

Set to a non−zero value if the target processor has an arithmetic coprocessor. Allows **idb** to recognize register names in expressions, the registers command to display the coprocessor registers and enables the **fregisters** command.

$count

Automatically set during the processing of **stop** and **when** breakpoints. Whenever the condition expression of a **stop** or **when** command is evaluated or the list of commands associated with a when command are executed, the value of **$count** is set to the number of times the **stop** or **when** breakpoint has occurred during the current execution of the target processor. All breakpoint counters are set to zero when the processor is started or continued.

The **$count** variable can, therefore, be used to cause an event to happen after a specified number of breakpoint occurrences:

```
stop at function when $count == 1000
```

will stop at the entry point of *function* when *function* has been entered 1000 times.

$currentline

Set to the line number of the current source line whenever the current line changes.

$disassyms

If set to a non−zero value, the **disassemble** command will not perform symbol lookups.

$exit

If **$exit** is set to a non−zero value, **idb** will set a breakpoint at an address corresponding to its value. When this breakpoint is encountered, idb will print ``Program terminated."

$instructions, $instructionshigh

These variables are set by targets that can do instruction counting. They represent the number of instructions that the processor has executed. **$instructions** is in the range 0..999,999,999 and **$instructionshigh** is incremented each time **$instructions** goes from 999,999,999 to 0.

The instruction counter is set to zero when the processor is reset.

$listline

Set to the line number last listed by the **list** and **displaysource** commands.

$listlength

Controls the number of source lines displayed by a **list** command in non−windowing mode.

$maxlatency, $curlatency

$maxlatency contains the maximum number of cycles that interrupts have been disabled since the processor was last reset and interrupts were enabled. **$curlatency** is the number of cycles that interrupts have currently been disabled.

$noclearall

Set to the maximum number of breakpoints that will be cleared in the target by individual ``clear breakpoints'' command. If a higher number of breakpoints exist, an emulator's ``clear all breakpoints'' command will be used, if it exists. If not, this variable has no effect.

$noexecute

When set to a non−zero value, *noexecute* mode is enabled.

When **idb** is in execute mode, it will set a breakpoint in the target using the target's hardware breakpoint mechanism for every user−requested breakpoint. Some targets, however, have one or few hardware breakpoints, or the target might execute the instruction on which the breakpoint is set, with undesired side effects.

In noexecute mode, **idb** writes a software interrupt instruction into memory at the requested breakpoint, and sets a single hardware breakpoint at the address **$returnpoint**. When the software interrupt is reached, an exception handler is executed that will copy the registers into memory, and execute a **nop** at **$returnpoint**. When the program is restarted, the registers are restored from memory, and the handler returns. The registers are written into memory indicated by the pointer whose address is **$savedsp**. If you manipulate the registers in *noexecute* mode, note that idb is manipulating the memory copy of the registers.

To use *noexecute* mode, make sure that *start.s* has been assembled with the symbol attach_traphandler defined in the *config.lib* file; this will attach the exception handler to the software interrupt.

The file *idbrc* contains the initialization of **$noexcecute** for the various target environments.

$pcrange

$pcrange is the address of a buffer in which an address pair is written that is assumed to be the bounds of execution when the byte value pointed to by **$rangeflag** is set to one.

$ramstart

Set to the beginning of the target program's read/write memory.

$ramend

Set to the end of the target program's read/write memory.

$rangeflag

$rangeflag, with **$pcrange**, affect how **idb** moves in programs with the step and next commands. When the two variables are set to non−zero values, user−supplied code is used to implement steps longer than one instruction in monitor−based targets. **idb** will write a range into the buffer specified by **$pcrange**, write the byte one into **$rangeflag**, and user−supplied code is expected to provide the step−in−range behavior.

$returnpoint

$returnpoint is the address to which **idb** sets its hardware breakpoint when running in *noexecute* mode.

$savedsp

When running in *noexecute* mode, **$savedsp** is the address of a pointer that points to the memory copy of the registers.

$showexec

Set to a non−zero value if executable line numbers should be marked with an asterisk in the source display.

$shownumber

Set to a non−zero value if the source line numbers should be shown in the source display.

$SIM05_MAXADDRESS

A mask that can be used to set the top of the 68HC05 simulator's address space. The simulator defaults to 0x1FFF. If this variable has a non−zero value when the simulator is reset, then the value is used as a mask for addresses. A typical value might be 0xFFFF, making the 68HC05 look like it has a full 64K address space.

$SIM12_PPAGE

This variable should be set to the window bank number of the window bank associated with the ppage register. Defaults to window bank 0.

$srcsyms

When set to a non−zero value, the disassembled code in a source window will not contain symbol names.

$threshold

When the debugger tries to convert a target system address into a symbol name, for example in the **where** or **disassemble** command, the address may or may not correspond directly to an object file symbol's address. **$threshold** is the maximum offset **idb** will use to print symbols by name; otherwise the symbol's hexadecimal address will be used.

Configuring idb targets

You must set the processor with the **−m** option and target types the **−t** option.

The −m option sets an idb variable called **$argm** which is used in the idbrc file to issue the appropriate processor command.

The −t option sets an idb variable called **$argt** which is used in the idbrc file to issue the appropriate targettype command.

Target types 16 and 333, 68HC16 and 683XX Background Debug Mode, are available only on MS−DOS host systems because **idb** requires access to an IBM PC−compatible parallel port.

imi

imi is the Introl monitor interface. It is an interface between your target hardware and **idb** which can be down−loaded or burned into EPROM. **imi**'s low−level commands are not suited for interactive use.

imi command set

g
: Restore the program's context and execute until encountering a breakpoint.

z
: Reload the stack pointer, disable interrupts, and restart the monitor.

w address count data
: Write *count* bytes of *data* to *address*. All arguments are assumed to be hexadecimal with no leading nor trailing identifying characters.

r address count
: Read *count* bytes of data from *address*. The data is output in hexadecimal, two characters per byte.

m source_address target_address count
: Copy *count* bytes of memory from *source_address* to *target_address*.

f begin_address end_address byte
: Fill memory from *begin_address* to *end_address* with *byte*.

Building imi

The source for **imi** is in $INTROL/Libraries/C/Introl_C/imi.c. You will need to examine the startup code, determine if the serial port functions in *imimon.c* are appropriate for your hardware, and determine your hardware's link mapping.

First, determine the memory configuration of your hardware, including interrupt vector table location (if appropriate), RAM location(s), ROM location(s), mapped hardware registers, etc. This information is necessary for configuring and building the startup code, and linking the monitor.

Enable *attach_traphandler* in *config.lib* when building the startup code for **imi**.

The next step is to determine whether or not **imi** contains the correct serial port code for your target. The relevant functions are defined separately and conditionally for each processor. The functions in *imimon.c* which need to be examined are:

open
: *open* initializes the serial port, which includes enabling the receiver and transmitter, and setting the baud rate.

EnableReceiverInterrupt
: This function enables the serial port's receiver interrupt. The handler to this interrupt is attached by a call to *_VECTOR* in *_ _main*. The name of the handler is *_ _idb_traphandler*, which causes **imi** to be re−entered when called.

_getchar
: This function receives one character from the serial port and return its value.

kbhit
: This function returns a non−zero value if a character is waiting to be read at the serial port.

_putchar
: This function outputs its integer argument to the serial port as a single character.

There are sets of these functions for every supported processor, so if you need to make changes, make sure you change the correct ones.

The following functions may also need to be modified:

_ _main
: This is the initialization entry point of **imi** which is called by the startup code. It installs the address of *imimon* into a global pointer called *_ _traphandler*, it attaches the exception handler in *trap.s* to the software interrupt vector and the serial port receiver interrupt, and it executes a software interrupt instruction. The code in *trap.s* depends on *_ _traphandler* to be set to the proper re−entry point. In addition, the calls to *_VECTOR* must use the proper vector numbers.

RERUN
: This function is called when the **z** command is processed. If you change the run−time environment, the code that loads the stack pointer might need to change.

Once these functions match your hardware, assemble the startup code:

```
asXX start.s
```

and copy *start.oXX* to the src directory. Then compile *imimon.c* with the command line:

```
ccXX imimon.c
```

Using the previously determined memory configuration information, modify a copy of the supplied linker command file *cXX.ld* to represent your specific environment. Then, link **imi** with the following command line:

```
ildXX cXX.ld start.oXX imimon.oXX −o imi
```

imi is now complete.

There are certain symbols which may be accessed by your program or by **idb** in i*dbrc* and must be exported from imi to your program. They are: *_ _savedsp*, *_ _vec*, and *_ _returnpoint*. To export them from **imi,** use the default **isym** command file *default.sym* located in the *src* directory to extract the symbols from **imi** with the following command:

```
isym imi
```

to produce *imisyms.ld* which is used while linking your programs to run under **imi**.

Once **imi** is running on your hardware, test all of its commands. A good test is the **z** command, which should reset **imi**, and display a prompt.

Imi−specific variables

The *idbrc* file contains code that sets a number of variables for **imi**. They are located in the *idbrc* file where the expression **$argt==1** appears. The following is a list of these variables and their function:

$entry
: **idb** uses the value of **$entry** as the program counter following a **reset** command.

$nobinary
: When set to zero, **idb** will send binary commands to **imi**. When set to 1, idb will send ASCII commands to **imi**. Binary commands are twice as fast as ASCII commands.

$useimistep
: When set to a non−zero value, **idb** will issue an s command to **imi** to perform a machine−level single step. Otherwise, **idb** performs single steps by decoding the current instruction and setting a breakpoint at its target. **imi** as supplied does not contain an s command; this facility exists so you can implement single step commands in your own imi−like interface to some executive.

Building programs for imi−based targets

Once **imi** is running on the target system, and tested, you can build a program to be run under the control of i**db/imi**. There are a few steps involved in doing this:

- Building the startup code
- Linking your program
- Running idb

First, the startup code must be re−configured so it does not interfere with **imi**. This requires having no vector table allocated in your programs. Change the *config.lib* file so that only the following options are set:

```
initialize_data
clear_uninitialized_data
```

and re−assemble the startup code. In order to link the program, modify your linker command file so that it maps your program to addresses that do not conflict with **imi**. Include *imisyms.ld* on the command line when linking your program.

You should now have a program that can be loaded into your target's memory space and run under the control of **idb/imi**. To run your program, run **idb** with its appropriate processor option and a target type of **−t1** for **imi**.

Background Debug Mode

bccremap_hook

This command procedure, defined in **idbrc**, is automatically executed when **idb**'s background mode driver starts or resets the processor. These functions can contain arbitrary code for configuring the target. By default, the functions configure the chip−select subsystem of the SIM identically to our supplied startup code. On power−up or reset, the SIM is reinitialized and must be configured. If you change the startup code, you must change the hook functions to match.

Basic background mode tips

If you are having trouble connecting via background mode with your target, try checking the following things first:

- Check to see if you're setting the port correctly.
- Check the cable connection.
- Try re−defining the **$bccremap_hook** function defined in the *idbrc* file so it does nothing, and then do a reset command. For more information, see the description of background mode.
- If there is a lot of noise or static in the area, try moving the target to a quieter place.
- Try a shorter cable.
- Try a different parallel port (or a different PC, if no other parallel ports are available).

Copyright