AMS

5 HSD Pattern Tools

When you save a pattern in the Digital Pattern Editor or Digital Pattern Debugger, the pattern in the display region is stored as a pattern file on disk. The pattern file has a `.pat` filename extension and the information in this file is in a binary code, known as "A500 pattern format."

To make the binary information in a pattern file available in ASCII format, IMAGE includes a set of digital pattern tools for translating a binary pattern file into an ASCII **IMAGE Pattern Language** (IPL) file and back again. Because IPL files are text files, the pattern information in these files can be readily exchanged with CAD tools or pattern files from other testers.

The IMAGE digital pattern toolset includes:

- An IMAGE Pattern Compiler (`ipc`) for translating IPL text files into binary pattern files (see page 5–17)
- An IMAGE Pattern Reverse Compiler (`iprc_hsd50`) for translating binary pattern files into IPL text files (see page 5–19)
- An IMAGE Pattern Beautifier (`ipb`) for making the contents of IPL text files more readable. It also prepares them for printing (see page 5–21).

5.1 IMAGE Pattern Language

An IPL file can include:

- A `pinmap_filename` statement
- Preprocessor statements
- External labels
- A `waveforms` statement
- `vector` statements
- An `output_filename` statement
- Comments

All IPL statements are in free format.

5.1.1 Pin Maps

A pinmap allows you to use pin numbers and pin names in IPL statements. Without a pinmap, you can only specify tester channels. Pinmaps must be in the correct form for Advanced Mixed-Signal (AMS) testers. (See "Pinmap Language" on page 4–6 in part I of the *Advanced Mixed-Signal Test Head Manual* for a description of the syntax.) Pinmaps must be specified in a separate file (if used) and passed to the IMAGE Pattern Compiler (`ipc`) using the following command (see "IMAGE Pattern Compiler" on page 5–17):

```
ipc -map <pinmapfile>
```

For example, the following is a valid pinmap file:

```
pinmap = {
1    "out_strobe"   dib:2   hsd50_drv:2,
2                   dib:5   hsd50_drv_rcv:5,
3                   dib:7   hsd50_drv_rcv:7,
4                   dib:9   hsd50_drv_rcv:9,
5                   dib:11  hsd50_drv_rcv:11,
};
```

You can also specify a pinmap file by adding a `pinmap_filename` statement to your IPL file. The syntax is:

```
pinmap_filename="<filename>";
```

This has the same effect as using the `-map <filename>` switch to `ipc`. This statement, if used, must precede the waveforms and vector statements. If both `-map` and the `pinmap_filename` statement are used, only the pinmap file specified by `-map` is read.

5.1.2 DIBView Schematics

DIBView schematics are like pinmaps. They allow you to use pin numbers and pin names in IPL statements. However, the pin numbers and names are specified in a DIBView schematic or `.exp` file (see "DIBView" on page 7–1 in the *IMAGE Base Language Manual*). DIBView schematics must be specified in a separate file and passed to the IMAGE Pattern Compiler (`ipc`) using the following command (see "IMAGE Pattern Compiler" on page 5–17):

```
ipc -map <dibview filename>.exp <pattern filename>.tp
```

5.1.3 Preprocessor Statements

The IMAGE Pattern Language supports C preprocessor statements such as `#define` and `#undef` for defining and undefining macros; `#include` for including definitions from another file; and `#if`, `#ifdef`, `#ifndef`, `#else`, and `#endif` for conditional compiling. The IMAGE Pattern Compiler passes all IPL source files through the C preprocessor (`cpp`) before compiling them. The following symbols are used in the descriptions of IPL syntax:

`|`	A vertical line represents `or`; you can choose one of the items separated by the vertical line.
`[ ]`	Square brackets indicate the enclosed expression is optional.
`...`	The preceding is repeated any number of times.
`#...#`	Any integer in the specified range (inclusive) can be used.

Pinmap files cannot be included with the `#include`. Pinmap files must be specified on the `ipc` command line with the `-map <pinmapfile>` switch or in the pattern source file using the `pinmap_filename` statement. Letters and names specified in upper case may be in either lower or upper case.

5.1.4 External Labels

Declaring external labels provides a way to refer to labels in another pattern. The HSD has two types of external pattern labels:

- Normal external labels
- PRAM-only external labels

The declaration type for normal external labels is `extern_label`. For example:

```
extern_label start2;  /* a label identifying the starting
                         location in another pattern file  */
```

```
extern_label subr2;   /* a label for a subroutine in
                                        another pattern file  */
```

The declaration type for PRAM-only external labels is `extern_pram_label`. The PRAM-only external label is used when referencing labels in routines compiled with the `-pram_only` switch (see "IMAGE Pattern Compiler" on page 5–17). This switch allows you to minimize PRAM usage by allowing CALL's to subroutines declared in this pattern to be placed in SAM. Patterns compiled with the `-pram_only` switch can only reference labels declared `-pram_only`. At compile time, the IMAGE compiler, `ipc`, checks for `extern_pram_label` if the compile `-pram_only` command is used. The compiler causes a run-time error if other labels are found.

Note *`-pram_only` generally works best for small pattern subroutines that are called many times. It is not recommended for use under any other conditions.*

For example:

```
extern_pram_label XXX;
extern_label YYY;
<many vectors>          /* may be in SAM */
CALL XXX;               /* may be in SAM */
<many other vectors>    /* the 1st 32 vectors here might be
                             in SAM                        */
CALL YYY;

<many other vectors>    /* the 1st 32 vectors here MUST be
                             in PRAM                       */
```

In this example, `ipc` knows that `XXX` is a PRAM-only label and `YYY` is not PRAM-only. Without a PRAM-only option, each `CALL` to a subroutine requires that the 32 vectors following the `CALL` opcode be placed in PRAM memory. However, if the `CALL` is to a PRAM-only subroutine, this restriction does not apply and the 32 vectors following the `CALL` to `XXX` might be placed in SAM. Other factors can cause some or all of the 32 vectors to be placed in PRAM whether `XXX` is PRAM-only or not. These factors are:

- Use of any opcode other than `CALL`, `REPEAT`, or `END_ARG`
- Use of `W`, `I`, `R`, or `C` in a vector
- A call to a subroutine which is not PRAM-only
- Use of any extended microcode instructions

If there are enough calls to `XXX`, the cost of forcing `XXX` to be in PRAM may be more than offset by the gain of putting the 32 vectors after each call to `XXX` into SAM instead of PRAM. Again, other factors may cause one or more these vectors to be in PRAM, so the gain is not automatic.

Calling a subroutine which is not PRAM-only requires that the next 32 vectors in the calling pattern be placed in PRAM because 32 vectors are needed to reset the SAM address counter. Since a normal subroutine may cause execution of both PRAM and SAM vectors before the `RETURN` occurs, the SAM address counter may have changed. So the 32 vectors following the `CALL` must be placed in PRAM. This allows the SAM address counter to be reset while they execute.

But if the SAM address counter can be guaranteed not to have changed since only PRAM vectors have been executed during the call to `XXX`, this restriction

does not apply and the vectors following the call may be placed in SAM if there is no other reason for them to be put in PRAM. If the subroutine is small and called often, this is likely to result in less PRAM usage overall.

External labels must be declared in a pattern before they can be used as references and must precede `vector` statements.

5.1.5 The waveforms Statement

If capture and source microcode commands are to be included in an IPL file, you must declare the source and capture memories for the commands. Use the `waveforms` statement to do this. The `waveforms` statement must appear before the first `vector` statement and has the following syntax:

```
waveforms={<pin specification> <instrument type>,...};
```

Where:

`<pin specification>` Is
`<pin name>`
`<pin number>`
`slot:<slot_number>`
`inst:<instrument_number>`

`<pin name>` Is the name of a pin specified in the pinmap. The pin must have a pin number in the pinmap if this form is used. If `dib` was used in the pinmap rather than a pin number (because the instrument is connected to DIB circuitry), then the `slot` or `inst` pin specifications must be used rather than a pin name.

`<pin number>` Is the number of a pin specified in the pinmap.

`<slot number>` Is the test head slot number of an analog waveform instrument. Slot numbers are not always portable to test systems with different configuration boards and should be used only when absolutely necessary (for example, while developing a test plan) and converted to pin names or pin numbers as soon as possible.

`<instrument_number>` Is the occurrence of a digital instrument in the test head. If there is only one of the instrument, this number is always 1.

`<instrument type>` Is one of the following:

`dig_cap`	Digital Capture Instrument
`dig_src`	Digital Source Instrument
`hfdig`	High Frequency Digitizer
`plfsrc`	Precision Low Frequency AC Source
`plfdig`	Precision Low Frequency AC Digitizer
`rt_histo`	Real Time Histogram Module
`vhfawg`	VHF Arbitrary Waveform Generator

Example:

```
waveforms = {
    LD      plfdig,     /* pin name                    */
    1       hfdig,      /* pin number                  */
    slot:4  plfsrc,     /* analog                      */
    inst:1  dig_src,    /* digital                     */
}
```

5.1.6 The vector Statement

Vector statements form the body of an IPL file. They begin with the keyword `vector` followed by a pin list and the vector data for the pins. Up to one million vectors can be specified in a single vector statement, and more than one vector statement can appear in an IPL file. (You should avoid large vector statements since large files compile slowly and are hard to debug. Breaking the large files into smaller files reduces the time needed to compile and debug.)

Like all IPL statements, `vector` statements have free format. The syntax for a `vector` statement is:

```
<vector statement>::= vector(<pin list>){<vector data>}
```

Pin List

A pin list assigns pins or channels to the vector data that follows the pin list. If the IPL file includes a pinmap, you can specify the pins as pin numbers or pin names. Or you can specify channel numbers.

You can group pins within a pin list by enclosing them within parentheses. Use a pingroup to pack together a number of pins, allowing the channel data to be specified for all of the pins within the pingroup. All pingroups in a pinlist must be separated by commas. Also individual items within a pingroup must be separated by commas.

Each pin or pingroup has a radix associated with it. You can assign a radix to each pin or pingroup in the pin list. When a radix is specified, the vector data for the pin or pingroup must assume this radix unless the pin list radix is overwritten by a local radix.

You can specify a mode for each group of pins. Legal modes are `D_D` (dual-drive), `IO_M` (io_midband), `IO_V` (io_valid) and `HIZ`. A mode indicates that the vector data for that pingroup is interpreted in a special way. The `D_D`, `IO_M`, and `IO_V` modes must have two data fields for each pingroup instead of the usual single data field.

If an IPL file contains more than one vector statement, all pins or pingroups specified in subsequent vector statements must be included in the first vector statement. The pins need not be in the same order. If you specify a mode for a pingroup, that pingroup cannot change modes between vector statements. Multiple vector statements are useful for specifying a pin or pingroup once (a clock, for instance) and then not repeating the data on subsequent vectors. Pins not specified in subsequent pin lists are set to “run time repeat” except for pins used with modes. Pins used with modes have special defaults for `Dual_Drive`, `IO_Valid`, `IO_Midband`, and `HiZ` as shown in table 5–1.

Table 5–1 Default Data for Unspecified Pins in Subsequent Pin Lists

Mode	Default Data for Unspecified Pins in Subsequent Pin Lists
D_D	0 0
IO_V	0 X
IO_M	0 X
HIZ	0

A `<pin list>` is defined as:

```
<pinlist>::= <pingroup>[:<modifier>][,] ...
```

Where:

```
<pingroup>::=   <field_name>
                <pin_name>
                <pin_number>
                <pin_number> TO <pin_number>
                CHAN:<digital_channel>
                CHAN:<digital_channel> TO
                     CHAN:<digital_channel>
                <field_group>
                .ISDN_DRV CHAN:<digital_channel>
                .ISDN_RCV CHAN:<digital_channel>

<field_group>::= ( <pin_name>,
                <pin_number>,
                <pin_number> TO <pin_number>,
                CHAN:<digital_channel>,
                CHAN:<digital_channel> TO
                CHAN:<digital_channel>, ...)

<modifier>::= <radix> [:<mode>]
            <mode> [:<radix>]
```

Where:

`<pingroup>`	Is any combination of pin names, pin numbers, or digital channel numbers separated by commas and enclosed in parentheses. For example: `(clk, 6 to 9, CHAN:15 to CHAN:20)`
`<field_name>`	Is a field name defined in a pinmap.
`<pin_name>`	Is a pin name defined in a pinmap. The pin must have a pin number in the pinmap if this form is used. If `dib` was used in the pinmap rather than a pin number (because the channel is connected to DIB circuitry), then the `CHAN:<n>` specification must be used instead of a pin name.
`<pin_number>`	Is a pin number defined in a pinmap, between 1 and 192.
`<digital_channel>`	Is the number for a digital channel. The number is between 1 and 192.
`<field_group>`	Is used to pack together several single pins. You can specify the data for all the pins at one time.
`<modifier>`	A modifier cannot be used with ISDN pins. If you use `<pin_number> TO <pin_number>` or `CHAN:<digital_channel> TO CHAN:<digital_channel>`,you must put either form in parenthesis before using a modifier.
`<radix>`	Is `X`, `H`, `Q`, `O`, `D`, `B`, or `S` `X` or `H` is hexadecimal `Q` or `O` is octal `D` is decimal `B` is binary `S` is symbolic (Default is symbolic.)
`<mode>`	Is `D_D`, `IO_V`, `IO_M`, or `HIZ`, where

D_D is dual-drive mode. Two drive data fields per pingroup per vector.

IO_V is io_valid mode. One drive and one expect data field per pingroup per vector.

IO_M is io_midband mode. One drive and one expect data field per pingroup per vector.

HIZ is high impedance mode. One drive data field per pingroup per vector.

Example:

```
vector ((1 to 3, 8, 9), 13, CHAN:9 to CHAN:5, input:D_D,
        (20 to 35):IO_M:H) { ...};
```

In this example, (1 to 3, 8, 9) is a pingroup. The numbers in this pingroup refer to pin numbers in a pinmap. Following the pingroup is the number 13, which refers to pin thirteen in a pinmap. CHAN:9 to CHAN:5 refers to digital channels five to nine in reverse order. input is the name of a pin defined in a pinmap. input is a dual_drive pin. (20 to 35) is another pingroup, referring to pins twenty to thirty-five in a pinmap. Pins twenty to thirty-five are io_midband mode and the vector data for this last pingroup must be specified in hexadecimal (:H).

Vector Data

Vector data specifies the pattern microcode and channel data associated with each pin or pingroup specified in a pin list. The format for the vector data is modeled after the Digital Pattern Editor format ("Creating a New Pattern" on page 3–25). As in the Digital Pattern Editor display, each vector data line represents a single vector in a digital pattern, and the vector data itself is grouped into columns as shown in figure 5–1. The information in these columns should be sufficient to reproduce the columns in the Digital Pattern Editor display.

```
VECTOR    VECTOR
NUMBERS   LABELS      PATTERN MICROCODE                CHANNEL DATA
{  0      GLOBAL p1:  TSET 1                           H1X01  .r0  .d1E00 .d0  .d0001 ;
   1                                                   L0X-1  .r0  .d1E01 .d0  .d0001 ;
   2                  SET_LOOP 10                      L1X-1  .r0  .d1E02 .d0  .d0001 ;
   3      loop1:                                       L0X-1  .r0  .d1E03 .d0  .d0001 ;
   4                  REPEAT 35                        L1X-1  .r0  .d1E04 .d0  .d0001 ;
   5                  END_LOOP loop1                   H0X-1  .r0  .d1E05 .d0  .d0001 ;
   6                  CALL subr 1                      L1X-1  .r0  .d1E06 .d0  .d0001 ;
   7      PLFDIG = (output) TRIG                       L0X-1  .r0  .d1E07 .d0  .d0001 ;
   8                  IF (PASS) CALL subr2             L1X-1  .r0  .d1E08 .d0  .d0001 ;
                                    •
                                    •
                                    •  }
```

Figure 5–1 Layout of Vector Data

All vector data for a pin list is enclosed within curly braces {}. Each line of vector data is terminated with a semicolon, which is interpreted by the pattern compiler as a boundary between two vectors. Vectors are further divided into the following fields (from left to right):

- Vector number field

- Vector label field
- Pattern microcode field
- Channel data fields

The fields correspond to cells in the Digital Pattern Editor. All fields are optional except for the channel data fields.

The fields themselves are grouped into columns, which correspond to the columns in the Digital Pattern Editor. Although the IMAGE Pattern Language imposes no restrictions on the format of the vector data, you should lay out the vector data in columns as shown in figure 5–1. (You can also have the IMAGE Pattern Beautifier do this for you "IMAGE Pattern Beautifier" on page 5–21.)

Vector Number Field

The syntax for a vector number is:

```
<vector_number>::=[+]<offset>[:<radix>]
```

Where:

`<offset>` Is an absolute vector offset from the beginning of the pattern. The first vector is always zero. The offset must be greater than the offset of the last defined vector and less than the maximum pattern size. Therefore, an `offset` of seven implies that this vector is the eighth vector from the beginning of the pattern. The offsets must always be increasing in size and less than the maximum pattern size.

Offsets need not be consecutive. One vector, for instance, can have an offset of 8 and the next vector an offset of 20. The pattern compiler assumes missing vectors are "runtime repeat" vectors unless the vectors have pingroups whose modes are specified. Defaults for pingroup modes are listed in table 5–2.

Table 5–2 Pingroup Mode Defaults

Mode	Default state
`D_D`	`0 0` (drive low - drive low)
`IO_V`	`0 X` (drive low - mask)
`IO_M`	`0 X` (drive low - mask)
`HIZ`	`0` (drive low)

`[+]<offset>` A plus sign before the offset changes the offset into a relative offset; that is, an offset relative to the current vector count. For example, `+3` repeats the current vector three times.

`[:<radix>]` If a radix is specified, that radix stays in effect until another radix is specified. (Default is decimal.)

Vector Label Field

The syntax for a vector label is:

```
<vector_label>::=[GLOBAL] <label>:
```

Where:

GLOBAL	Is a keyword for making this label a global label. (Global labels are explained in "Pattern Microcode" on page 2–36.)
<label>	Must be a legal C identifier; that is, an alphanumeric string with the first character being an alphabetical character. Underscores are accepted as alphabetical characters.

Pattern Microcode Field

A pattern microcode field has three optional commands that must appear in the order shown:

```
<MICROCODE COMMAND> <SOURCE/CAPTURE COMMAND> <TSET COMMAND>
```

The four types of MICROCODE COMMANDS are:

- Unconditional commands
- Conditional commands
- Conditional statements
- Other commands

(Microcode commands are covered in detail in "Pattern Microcode" on page 2–36)

Unconditional commands include:

SET_LOOP <1..65536>	Push number onto loop stack. SET_LOOP 1 executes vectors in loop once.
SET_LOOP1 <1..65536>	Set loop counter 1 to number. No loop stack for loop counter 1.
SET_LOOP2 <1..65536>	Set loop counter 2 to number. No loop stack for loop counter 2.
LOOP <1..65536>	Similar to SET_LOOP, but pushes number onto the stack the first time only, not each time the loop executes.
LOOP1 <label>	Similar to SET_LOOP1, but sets loop counter 1 the first time only, not each time the loop executes.
LOOP2 <label>	Similar to SET_LOOP2, but sets loop counter 2 the first time only, not each time the loop executes.

Note *LOOP, LOOP1, and LOOP2 instructions work as described above only if the LOOP, LOOP1, or LOOP2 instruction is at a label, and the corresponding END_LOOP references that label. See "LOOP, LOOP1, and LOOP2" on page 2–41.*

END_LOOP <label>	Decrement loop counter. If not zero go to <label>, otherwise pop loop stack.
END_LOOP1 <label>	Decrement loop counter 1. If not zero go to <label>, otherwise continue.
END_LOOP2 <label>	Decrement loop counter 2. If not zero go to <label>, otherwise continue.
REPEAT <2..32768>	Repeat channel data <2..32768> times before continuing.

POP_LOOP	Pop the loop stack.
PUSH <label>	Push address of <label> onto subroutine stack.
HALT	Halt pattern.
READCODE <0..2047>	Set read code to <0..2047>.
CLR_CODE	Clear the read code.
SET_GLO <label>	Put address of <label> in global address register.
MATCH <label>	If fail and not end of loop go to <label>, otherwise continue.
ENABLE <[AND | OR] ([!] <flag>)>	Set up conditions to be evaluated by a later IF <flag> statement. The flag can be: PASS | FAIL | EXT | SCF | CPU | NONE. NONE means FLAG always evaluates false (no conditional action occurs).

Note *AND and OR are legal only when used with ENABLE command. The AND or OR is required if more than one condition is specified.*

CLR_FLAG (<flag>...)	Clear the specified condition flags if they were previously enabled.
POP	Pop the subroutine stack.
KEEP_ALIVE	Activate keep-alive RAM.
SET_SCF	Set the SCF flag on the other SCM.

Conditional commands include:

JUMP <label>	Go to <label>.
CALL <label>	Execute subroutine at <label>.
RETURN	Return from subroutine.
END_ARG	End an argument list in a subroutine.
EXE_GLO	Push address of next vector onto stack, then jump to address in global address register.
JMP_GLO	Start execution at address in global register.
EXIT_LOOP <LABEL>	Pop loop stack, go to <label>.

A conditional statement has the form:

```
IF (FLAG) <conditional command> [CLR_COND]
```

The condition flags are:

FLAG	Evaluate conditions programmed by previous ENABLE statement.
PASS	Inverse of FAIL flag.
NONE	Clear any ENABLE'd conditions.
FAIL	Becomes true when a failure on any channel gets back to the SCM. Remains true until cleared by the pattern.

EXT	Vector bus or formatter condition true.
SCF	Dual SCM flag true.
CPU	CPU flag is true. (This flag is set when a `resume hsd50 pattern` statement is executed in a test program.)
!EXT	Vector bus or formatter condition false.
!SCF	Dual SCM flag false.
!CPU	CPU flag is false.

Other commands include:

NO_HALT	Do not stop the pattern if a failure occurs on this vector.
ICYC	Inhibit cycle counter.
RESYNC	Synchronizes T0 clock with AC cage clocks. RESYNC, A0_INC, and A0_DEC are mutually exclusive.
MASK	Mask failures on this vector
CLR_FAIL	Clear the formatter accumulated fail information.
QUAL	Causes the specified vector to be captured in HRAM when hram_mode:vectors is specified.
A0_INC	Increments A0 divider. RESYNC, A0_INC, and A0_DEC are mutually exclusive.
A0_DEC	Decrements the A0 divider. RESYNC, A0_INC, and A0_DEC are mutually exclusive.
CLR_COND	Clears the condition flags used to make conditional branches in the pattern. Only flags currently enabled are cleared, and they are cleared only if a branch occurs. This instruction must be used with a conditional command.

A SOURCE/CAPTURE COMMAND can either be a source instrument command or a capture instrument command. Source instrument commands come in two forms:

```
<dig_src keyword> = (<analog spec.>) [<dig_src control>] [VBC_STR]
<dig_src keyword> = (<digital spec.>) [<dig_src control>] [SEND | SEND10 |
                          SHIFT] [VBC_STR]
```

Capture instrument commands also come in three forms:

```
<c_mem keyword> = (<analog spec.>)[TRIG][VBC_STR]
<c_mem keyword> = (<digital spec.>)[TRIG][STORE][SHIFT][VBC_STR][RESYNC]
RT_HISTO = (<digital spec.>)[TRIG][STORE][SHIFT][VBC_STR] [DECR]
```

Where:

<dig_src keyword>	Is one of the following:
DIG_SRC	Digital Source instrument.
PLFSRC	Precision Low Frequency AC Source.
VHFAWG	VHF Arbitrary Waveform Generator.

<analog spec.>	Is the name or number of an analog pin defined in your pinmap, or it is SLOT:<slot number> where slot number is the number of the slot in the advanced mixed-signal test head where the desired instrument is located. This form is not portable to test systems with differing configuration boards and channel card populations and should not be used unless necessary.
<digital spec.>	Is the name or number of a digital pin defined in your pinmap or it is: INST:<instrument number> where instrument number is between 1 and N. N is the number of instruments in the test system.
<dig_src control>	Is one of the following:
RESYNC	Resynchronize clocks on the VHFAWG. This command can be used on the VHFAWG only.
START	Start sourcing immediately and loop continuously. Can not be used with the VHFAWG.
STARTE	Start sourcing at the end of the current waveform and loop continuously.
START1	Start sourcing immediately. Source the data once and then stop. Cannot be used with VHFAWG.
STARTE1	Start sourcing at the end of the current waveform. Source the data once and then stop. Cannot be used with VHFAWG.
NEXT	Start sourcing the next waveform after the current waveform is done. Then loop continuously.
NEXT1	Start sourcing the next waveform after the current waveform is done. Source the data once and then stop. Can not be used with the VHFAWG.
STOP	Stop sourcing immediately. Can not be used with the VHFAWG.
STOPE	Stop sourcing at the end of the current waveform.
<c_mem keyword>	Is one of the following: DIG_CAP Digital Capture Instrument HFDIG High Frequency AC Digitizer PLFDIG Precision Low Frequency AC Digitizer

A TSET COMMAND defines the timing set for the channel data. Its syntax is:

TSET <1..1023>

Channel Data Field

Each vector has channel data associated with it. Like the vector itself, the channel data for each vector is divided into fields, which are organized into columns. Each channel field corresponds to a pin or pingroup in the pin list. The

pin list determines the number of channels for the channel data and the number, order, and radix of each channel field.

The number of channels in a channel field is determined by the number of pins in the pingroup, and the radix of a channel field is determined by the radix of the pingroup. Any single pins in a pin list have channel fields consisting of a single channel.

The syntax for the channel fields depends on their radix. The syntax is:

```
<field data> = <pin_value>[<pin_value>...]
               | <numeric_value>[:<radix>]
               | <symbolic_value>
               | <isdn_data>
```

Where:

`pin_value` — Is `-` | `0` | `1` | `L` | `M` | `H` | `V` | `X` | `W` | `I` | `R` | `C`

`numeric_value` — Is `.D<number>` | `.R<number>`

`.D<number>` — Indicates that the channels are to be driven with the specified `<number>`.

`.R<number>` — Indicates that channels are to receive data which are to be compared against the following `<number>`.

`number` — Represents the drive or receive states for the channels. How they are defined depends on which numeric radix is specified. For example, in binary the `<channel states>` for a field of eight drive channels might be defined as `.d10011111`. In hex they would be defined as `.d9F` and in octal they would be defined as `.d237`.

`symbolic_value` — Is:

`0`	Drive low
`.1`	Drive high
`.L`	Expect low
`.M`	Expect midband
`.H`	Expect high
`.V`	Expect valid
`.W`	Waveform drive. Data comes from waveform memory
`.I`	Waveform drive inverted
`.R`	Waveform receive
`.C`	Waveform receive inverted
`.X`	Tri-state
`.-`	No change

`isdn_data` — Is `<isdn_drv_dat>` | `<isdn_rcv_dat>`

Note *`isdn_data` is legal only for pins declared `.ISDN_DRV` or `.ISDN_RCV` in the pinlist.*

`isdn_drv_dat` — Is:
`W` | `X` | `-` | `HOLD` | `1` | `0` | `0H` | `0L` | `E` | `B` | `1_VIOL` | `0_VIOL` | `0H_VIOL` | `0L_VIOL` | `E_VIOL` | `B_VIOL` | `1_BT` | `0_BT` | `0H_BT` | `0L_BT` | `E_BT` | `B_BT` | `1_BTV` | `0_BTV` | `0H_BTV` | `0L_BTV` | `E_BTV` | `B_BTV`

isdn_rcv_data	Is: HOLD | X | - | H | L | LH | LL | M | V | B | H_VIOL | L_VIOL | LH_VIOL | LL_VIOL | M_VIOL | V_VIOL | B_VIOL | H_LE | L_LE | LH_LE | LL_LE | M_LE | V_LE | B_LE | H_VE | L_VE | LH_VE | LL_VE | M_VE | V_VE | B_VE | H_CF | L_CF | LH_CF | LL_CF | M_CF | V_CF | B_CF

A hyphen (-) is used to repeat symbolic channel data. The hyphen means "no change," which repeats the channel data from the previous vector. For numeric channel data, a period-hyphen (.-) represents no change. Also, a period before a single symbolic digit assigns that symbolic digit to all channels in a pingroup.

For example, H1X-1 is a symbolic channel field represents its channel data as a series of symbolic digits, one for each channel.

5.1.7 The output_filename Statement

By default, the pattern compiler gives its output file the same root name as its source file but appends onto it a .pat extension. For example, if the IPL source file is named bonzo.tp, the compiler's output file is automatically named bonzo.pat.

You can override this default action by adding an output_filename statement to your IPL file. Its syntax is:

```
output_filename="<filename>";
```

When the pattern compiler detects an output_filename statement, it uses the filename supplied in the statement for the name of its output file. Specifying the name of the output file is useful for conditional compiles where several pattern files are produced from one IPL source file. This statement, if used, must precede the first vector statement in an IPL file.

5.1.8 Comments

Use the normal C syntax (/* */) to include comments in an IPL file. The pattern compiler preserves these comments in its output file, so that they appear again in a pattern editor or pattern debugger display unless the -nocomment switch is used with the ipc command (see "IMAGE Pattern Compiler" on page 5–17). However, it does not preserve two types of comments. Any comment starting with either /*@ or /*@@ is discarded by the compiler. These comments never appear in a pattern editor or pattern debugger display. For example:

```
/* This comment appears in the pattern editor. */
/*@ This comment does not appear in the pattern editor.*/
```

Like the pattern compiler, the pattern beautifier preserves all comments specified in normal C syntax (/* */). It also preserves all comments that start with /*@. But it discards all comments that start with /*@@.

The pattern compiler associates any comments it finds with the vector currently being compiled. For instance, if it finds a comment at the end of a vector line, it associates it with that vector. If it finds a comment between two vector lines, it associates it with the second vector. If one vector has multiple comments, the comments are grouped together as a single comment in an output pattern file.

5.1.9 IPL Reserved Words

IMAGE Pattern Language (IPL) has a separate set of reserved words that is different from IMAGE Test Language (ITL). One key difference between IPL reserved words and ITL reserved words is that IPL is *not* case-sensitive. So if VHFAWG is reserved, all of vhfawg, Vhfawg, VhFaWg, and every other spelling of VHFAWG are also reserved.

Words used in pattern syntax are reserved. See sections 2, 3, and 4 for the syntax for pattern files. See the "Reserved Word List" on page A–1 of the *IMAGE Base Language Manual* for reserved words in IPL and ITL.

5.1.10 Sample IPL File, Including Separate Pinmap File

```
/* Pinmap file: examplemap.h */
pinmap = {
1   "out_strobe"    dib:H50_1       hsd50_drv:1,
2   "clock"         dib:H50_2       hsd50_rcv:2,
3   "aux"           dib:H50_3       hsd50_rcv:3,
4   "output"        dib:105         plfsrc_hi,
5   "output_rtn"    dib:106         plfsrc_lo,
6   "input"         dib:H50_4       hsd50_rcv:4,
7   "VDD"           dib:107         dutsrc,
8   "enable1"       dib:H50_8       hsd50_rcv:8,
9   "enable2"       dib:H50_9       hsd50_rcv:9,
10                  dib:H50_10      hsd50_drv:10,
11                  dib:H50_11      hsd50_drv:11,
12                  dib:H50_12      hsd50_drv:12,
13                  dib:H50_13      hsd50_drv:13,
14                  dib:H50_14      hsd50_drv:14,
15                  dib:H50_15      hsd50_drv:15,
16                  dib:H50_16      hsd50_drv:16,
17                  dib:H50_17      hsd50_drv:17,
18                  dib:H50_18      hsd50_drv_rcv:18,
19                  dib:H50_19      hsd50_drv_rcv:19,
20                  dib:H50_40      hsd50_drv_rcv:40,
21  "GROUND"        dib:108         dutsrc,
22                  dib:H50_35      hsd50_drv:35,
23                  dib:H50_34      hsd50_drv:34,
24                  dib:H50_33      hsd50_drv:33,
25                  dib:H50_32      hsd50_drv:32,
26                  dib:H50_31      hsd50_drv:31,
27                  dib:H50_30      hsd50_drv:30,
28                  dib:H50_29      hsd50_drv:29,
29                  dib:H50_28      hsd50_drv:28,
30                  dib:H50_27      hsd50_drv:27,
31                  dib:H50_26      hsd50_drv:26,
32                  dib:H50_25      hsd50_drv:25,
33                  dib:H50_24      hsd50_drv:24,
34                  dib:H50_23      hsd50_drv:23,
35                  dib:H50_22      hsd50_drv:22,
36                  dib:H50_21      hsd50_drv:21,
37                  dib:H50_20      hsd50_drv:20,
38                  dib:H50_36      hsd50_rcv:36,
39                  dib:H50_37      hsd50_rcv:37,
40                  dib:H50_38      hsd50_rcv:38,
41                  dib:H50_39      hsd50_rcv:39,
42  "VCC"           dib:109         dutsrc,

(10,11,12,13,14,15,16,17)                           "databyte"  field,
(18,19,20)                                          "iocode"    field,
(37,36,35,34,33,32,31,30,29,28,27,26,25,24,23,22)   "byteaddr"  field,
```

```
(38,39,40,41)                                         "iocontrol" field,
    };  /* end of pinmap file: examplemap.h */

/* Source File: example.tp */
waveforms = {  input  PLFSRC,  output PLFDIG }
vector ((1 to 3, 8, 9), databyte:D, byteaddr:H, iocode:O, iocontrol:B) {
/*@
*LABEL       COMMAND                 12389    databyte byteaddr   iocode   iocontrol */
GLOBAL p1:  TSET 1                  H1X01    .r0      .d1E00      .d0      .d0001 ;
                                    L0X-1    .r0      .d1E01      .d0      .d0001 ;
            SET_LOOP 10             L1X-1    .r0      .d1E02      .d0      .d0001 ;
LOOP1:                              L0X-1    .r0      .d1E03      .d0      .d0001 ;
            REPEAT 35               L1X-1    .r0      .d1E04      .d0      .d0001 ;
            END_LOOP loop1          H0X-1    .r0      .d1E05      .d0      .d0001 ;
            CALL subr1              L1X-1    .r0      .d1E06      .d0      .d0001 ;
            PLFDIG = (output) TRIG  L0X-1    .r0      .d1E07      .d0      .d0001 ;
            IF (PASS) CALL subr2    L1X-1    .r0      .d1E08      .d0      .d0001 ;
            HALT ICYC PLFSRC = (input) STOP
                                    L0X-1    .r0      .d1E09      .d0      .d0001 ;
subr1:      READCODE 101            L1X-1    .r127    .d1E0a      .d0      .d0001 ;
            PLFSRC = (input) VBC_STR
                                    L0X-1    .r126    .d1E0b      .d0      .d0001 ;
            PLFSRC = (input) TRIG   L1X-1    .r125    .d1FFF      .d0      .d0001 ;
            RETURN                  L0X-1    .r124    .d1FFF      .d0      .d0001 ;
subr2:      SET_GLO j1              L1X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    H0X-1    .r0      .d1FFF      .d0      .d0001 ;
            IF (EXT) JMP_GLO        L1X-1    .r0      .d1FFF      .d0      .d0001 ;
            REPEAT 5                L0X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    L1X-1    .r0      .d1FFF      .d0      .d0001 ;
j1:         RETURN                  L0X-1    .r0      .d1FFF      .d0      .d0001 ;
GLOBAL p2:  TSET 5                  L1X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    L0X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    L1X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    L0X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    L1X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    L0X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    L1X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    L0X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    L1X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    L0X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    H1X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    L0X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    L1X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    L0X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    L1X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    L0X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    L1X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    L0X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    L1X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    L0X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    L1X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    L0X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    L1X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    H0X-1    .r0      .d1FFF      .d0      .d0001 ;
                                    L1X-1    .r0      .d1FFF      .d0      .d0001 ;
}
```

5.2 IMAGE Pattern Compiler

The IMAGE Pattern Compiler (ipc) compiles IMAGE Pattern Language (IPL) files containing vector data and produces a binary pattern file (a file with the .pat extension) suitable for loading into tester memory or for examination or modification by the pattern editor. ipc can produce pattern files for A500 hardware or AMS (Advanced Mixed-Signal) hardware. It determines the targeted hardware type in the following way:

- If the -filetype <hardware type> switch to ipc is used, it produces pattern files targeted for that hardware type. Legal types are a500 and hsd50.
- If no -filetype switch is found, the .tp file containing the pattern data is scanned for a filetype = <hardware type> statement. This statement must occur before the first vector statement in the file. Legal hardware types are a500 and hsd50.
- If neither a -filetype switch nor a filetype statement is found, the targeted hardware type is the A500.
- If both a -filetype switch and a -filetype statement are found and they are different, the -filetype switch overrides the -filetype statement. A warning is issued if this occurs.

ipc runs fastest when the -filetype <hardware type> switch is used, since the file need not be read to determine the target hardware type. Invoke it using the following command:

```
ipc [ -filetype <hardware type>          ]  <infile>
    [ -output <output file>              ]
    [ -define <name>                     ]
    [ -define <name>=<def>               ]
    [ -map <pinmapfile or DIBViewfile>   ]
    [ -max_errors <count>                ]
    [ -tab <tab width>                   ]
    [ -nocomments                        ]
    [ -pram_only                         ]
    [ -pram_size <number>                ]
    [ -compat                            ]
    [ -1m                                ]
    [ -scan <scan memory type>           ]
    [ -no_delimiters                     ]
```

Where:

-filetype <hardware type>
Specifies the target hardware type. Legal types are a500 and hsd50.

<infile>
Is the name of one or more IPL files. If you do not specify a filename extension, the pattern compiler assumes it has the extension .tp. For instance, the command ipc demo is interpreted as ipc demo.tp.

-output <outfile>
Specifies the name of the output file. By default, ipc gives its output file the same root name as its source file, but appends onto it a .pat filename extension. This switch allows you to specify a different name for the output file.

`-define <name>`	Defines `<name>` as the number one (1). Used for conditional compiling.
`-define <name>=<definition>`	Replaces every occurrence of `<name>` in the source file with its `definition`.
`-map <pinmap file, DIBView file>`	Is the name of a file containing a pinmap or DIBView schematic.
`-max_errors <count>`	Is the number of errors to allow before aborting the compile. The default is 50.
`-tab <tab width>`	When the pattern compiler encounters tabs in its source file, it converts them to spaces in its output file. `tab` specifies the number of spaces for each tab. This switch can be used to preserve the readability of comments. (Default is 8)
`-nocomments`	Prevents comments from being included in the output file. This decreases the size of the output file.
`-pram_only`	Directs the pattern compiler to compile the pattern for PRAM-only. The pattern is placed entirely in PRAM instead of being split between PRAM and SAM. This switch works best for small pattern subroutines called many times and is not recommended for any other use. This switch is for AMS patterns only.
`-pram_size <number>`	`ipc` assumes a PRAM size of the specified `number`.The default is 16k (16384). This switch is for AMS patterns only.
`-compat`	`ipc` accepts certain syntax that is legal on A500 but not normally accepted for the AMS.
`-1m`	Using the `-1m` switch creates patterns with the `unrestricted split`. `ipc` allows you to compile patterns with the `restricted` (SAM/PRAM) `split` (the default state) or `unrestricted split`. Use `unrestricted split` only when compiling patterns for test systems containing DMF boards having revisions -07 or higher. Pattern created with `restricted split` can run on any testers, but the patterns do not make full use of SAM memory due to a hardware problem corrected in DMF boards with revisions -07 and higher (see "Digital Pattern Debugger" on page 4–1).
`<scan memory type>`	Is `pramsam` – selects the parallel memory `vbms` – selects the VBMS If no `-scan` flag is used, the default is `pramsam`.
`-no_delimiters`	This switch removes comment delimiters (`/*` and `*/`) from the pattern file during compilation. Delimiters do not appear in the pattern editor or debugger. The reverse compiler (`iprc_hsd50`) restores the comment delimiters when creating the `.tp` file,

although the spacing may be slightly different than in the original .tp file.

Examples:

```
ipc demo
```

The source file is named demo.tp, which is an IPL file. For demo.tp, the pattern compiler generates a binary pattern file named demo.pat. Another example is

```
ipc -define version_a demo
```

Again, the IPL source file is named demo.tp and the output file is named demo.pat. Any statement between #ifndef version_a and #endif are discarded. Any statement between #ifdef version_a and #endif are included in the compilation.

```
ipc -nocomments -output toad.pat demo
```

The IPL source file is named demo.tp. During compilation, the pattern compiler discards all comments in demo.tp. The output from the compilation is stored in a pattern file named toad.pat.

5.3 IMAGE Pattern Reverse Compiler

The IMAGE Pattern Reverse-Compiler (iprc_hsd50) reverses the actions of the pattern compiler. It takes the contents of a binary pattern file (.pat), converts it to IPL syntax, processes the IPL code through the IMAGE Pattern Beautifier (ipb), then outputs it to an IPL text file.

The command for reverse-compiling a pattern is:

```
iprc_hsd50 | -map <pinmap file or        | <pattern file(s)>
           |     DIBView file>           |
           | -f                          |
           | -include <infile(s)>        |
           | -output <outfile>           |
           | -define <name>              |
           | -define <name> = <def>      |
           | -noipb                      |
           | -tab <number>               |
           | -addvectnum                 |
           | -header                     |
           | -width <number>             |
           | -length <number>            |
           | -page                       |
           | -oneline                    |
```

Where:

-map <pinmap file or DIBView>
: Specifies the name of the file containing the pinmap or DIBView schematic.

-f
: This forces the recompile. It overwrites the existing output file without confirmation.

-include <infile(s)>
: Adds the specified #include statement to the output file.

-output <outfile>
: Specifies the name for the output file. By default, the reverse compiler is given the same root name as the pattern file with the filename extension changed to

	`.tp`. This switch allows you to override this default and specify any output file name. If the name does not have a `.tp` extension, the pattern beautifier rejects it.
`-define <name>`	Define `<name>` as the number one (1) wherever it occurs in the pinmap. Used when reverse-compiling a pattern which has a conditionally compiled pinmap.
`-define <name> = <def>`	Replaces every occurrence of `<name>` in the pinmap with `<def>`. Used when reverse-compiling a pattern which has a conditionally compiled pinmap.
`-noipb`	Inhibits the pattern beautifier from processing the output file.
`-tab <number>`	Directs the pattern beautifier to replace each tab with the specified number of spaces.
`-addvectnum`	Directs the pattern beautifier to add a vector number to each vector.
`-header`	Directs the pattern beautifier to include column headers in its output, similar to what you would see in a pattern editor display.
`-width <number>`	Tells the pattern beautifier the page width for the output file.
`-length <number>`	Tells the pattern beautifier the number of lines per page for the output file.
`-page`	Directs the pattern beautifier to break the pages at page boundaries. If the `-header` switch is also specified, column headers are included at the top of each page.
`<pattern file(s)>`	Identifies the name of the pattern file or files for the reverse-compilation. If you do not specify an extension in the file name, `.pat` is assumed.
`-oneline`	Directs the pattern beautifier to put all data from a single vector input line onto the same output line, even if the line width is exceeded. If this switch is not specified, `ipb` starts a new line when the line width is exceeded. (The default line width is 132 characters, but the default can be changed using the `-width` switch.)

5.3.1 Pinmaps and DIBView Schematics

A pinmap or DIBView schematic allows the reverse-compiler to refer to DUT pins directly through the pin numbers and pin names defined in the pinmap or DIBView schematic rather than indirectly through tester channel numbers. The reverse-compiler must be notified that the pinmap file or DIBView schematic exists. Do this using the `-map` switch in the `iprc_hsd50` command. The reverse-compiler responds to `-map` by extracting the pinmap or DIBView schematic from the named file and using it to map tester channels to DUT pins when defining its pin list.

5.3.2 Specifying the Output File

By default, the output file name is given the same root name as the pattern file with the filename extension changed to `.tp`. The `-output` switch allows you to override this default and specify any output file name. If the output file already exists, you are asked to confirm the overwrite unless the `-f` switch is specified.

5.3.3 Specifying Include Files

The reverse-compiler has no knowledge of what `#include` files were used when compiling a pattern. Therefore, you must specify any `#include` files using the `-include` switch in the `iprc_hsd50` command. For each file specified in the `-include` switch, the reverse compiler generates an `#include` statement in its output file, in the order listed in the switch.

5.3.4 Beautifying the Output

To make the IPL code more readable, the reverse-compiler automatically sends its output to the pattern beautifier for processing. The switches `-tab`, `-addvectnum`, `-header`, `-width`, `-length`, and `-page` are all passed to the beautifier to customize the output.

Running the output text from the reverse-compiler through the pattern beautifier adds extra time to the translation process. If the appearance of the output is not important, you can reduce processing time by not running the output through the beautifier. Use the `-noipb` switch to bypass the pattern beautifier. Specifying `-noipb` deactivates `-tab`, `-addvectnum`, `-header`, `-width`, `-length`, and `-page`.

5.3.5 Comments

The reverse-compiler automatically scans comments in a pattern and inserts missing comment delimiters (`/*` and `*/`) where necessary. This ensures that the IPL file produced by the reverse-compiler can be recompiled by the pattern compiler without errors or modification.

5.3.6 Limitations of the Reverse Compiler

If a pattern is compiled and then reverse-compiled, the resulting pattern may not match the original pattern. This is because some features of the IMAGE Pattern Language are not recoverable from the binary pattern (`.pat`) file. The limitations are as follows:

- Any C preprocessor directives (such as `#define`, `#ifdef`, and `#include`) are lost, except for the include files specified in an `iprc_hsd50` command.
- Channel and pin specifications may not appear exactly as originally defined. The rules are:
  - If `-map` is specified in the `iprc_hsd50` command and the channel is in the pinmap, the reverse-compiler uses the DUT pin name. If it has no name, the reverse-compiler uses the pin number.
  - If there is no pinmap or the channel is not found in the map, the reverse-compiler uses the `INST: or SLOT:` syntax and the instrument or slot number.
- If a pin is mapped to more than one analog instrument, only the first instrument is mapped to the pin.

5.4 IMAGE Pattern Beautifier

The IMAGE Pattern Beautifier (`ipb`) makes the contents of an IPL file more readable by organizing the vector fields into columns. It can also prepare the text for printing by breaking it into pages of specified length and width with column headers on each page. In addition, the pattern beautifier can add vector numbers to each vector or remove them from each vector. Beautifying an IPL file does not affect the way it compiles.

The pattern beautifier does not always correctly beautify an IPL file containing errors. However, it does not introduce errors either.

The command for beautifying an IPL file is:

```
ipb [ -width <number>                ] <input file>
    [ -page [-length <number>]       ]
    [ -header                        ]
    [ -tab <number>                  ]
    [ -addvectnum | -stripvectnum    ]
    [ -inplace | -output <outfile>   ]
    [ -oneline                       ]
```

Where:

`-width <number>`	Specifies the page width in columns. (Default is 132 columns)
`-page`	Breaks the text into pages and prints a new header on each page if `-header` is specified.
`-length <number>`	Specifies the number of lines per page. (Default is 60)
`-header`	Includes column headers, similar to what you would see in a Digital Pattern Editor Display.
`-tab <number>`	Specifies the number of spaces for each tab. A tab size of 0 (zero) means no tabs. (Default is 8)
`-addvectnum`	Adds a vector number to each vector.
`-stripvectnum`	Removes all vector numbers.
`-inplace`	Writes the output back to the `<input file>`, while saving the original file as `<input file>%`.
`-output <outfile>`	Writes the output to `<outfile>` instead of displaying it on your terminal (standard output).
`<input file>`	Is the name of the input file for the pattern beautifier. If the name does not include a filename extension, `.tp` is assumed.
`-oneline`	Directs the pattern beautifier to put all data from a single vector input line onto the same output line, even if the line width is exceeded. If this switch is not specified, `ipb` starts a new line when the line width is exceeded. (The default line width is 132 characters, but the default can be changed using the `-width` switch.)

An example would be:

```
ipb -width 80 -page -header -tab 0 -addvectnum -inplace
myfile
```

or

```
ipb -w 80 -p -h -t 0 -a -i myfile
```

This command beautifies the file `myfile.tp`. In the process, it arranges the output to fit on an 80 space by 60 line page with a header comment at the beginning of each page. No tabs are used. Each vector is preceded by a vector number. The output is written back to the file `myfile.tp`.

5.4.1 Comments

The pattern beautifier normally preserves comments. If the `-header` option is specified, the header information is included as a comment at the beginning of a vector statement. If `-page` is specified with the `-header` switch, a header is added to each new page.

When the pattern beautifier adds a comment, it begins the comment with `/*@@`. But it also discards any comments in the input file beginning with `/*@@`. This means the pattern beautifier can process an IPL file a second time without reproducing its comments a second time.

All comments within vector statements are repositioned, if necessary, on a line (or lines) by themselves, between the ending semicolon of one vector and the beginning of the next vector.

5.4.2 Vector Numbers

By default, the pattern beautifier preserves all vector numbers. To remove vector numbers from an IPL file, specify the `-stripvectnum` switch. `stripvectnum` does not affect vector numbers specified as relative offsets (vector numbers preceded by "+").

The `-addvectnum` switch adds vector numbers to the beginning of any vector that does not already have one.

5.4.3 Example

Given the following IPL text:

```
vector ((1 to 3, 8, 9), 6, databyte:D, byteaddr:H, iocode:O, iocontrol:B,
(43 to 58):H, 59, 60, 61, 62, 63)

 {
 GLOBAL p1: TSET 1 H1X01 L .r255 .d1E00 .d0 .d0001 .r1234 H 0 1 L H;
loop1: LOOP 10 L1X-1 0 .r253 .d1E02 .d755 .d0001 .r5678 0 1 L H 0 ;

L0X-1 1 .r252 .d1E03 .d0 .d0001 .dBA98 1 L  H 0 1 ;
REPEAT 35 L1X-1 L .r251 .d1E04 .d0 .d0001 .r1234 L H  0 1 L;
END_LOOP loop1 H0X-1 H .r250 .d1E04 .d0 .d0001 .r1234 L H 0 1 L;
CALL subr1 L1X-1 0 .r249 .d1E06 .d0 .d0001 .r5678 0 1 L H 0;
```

The pattern beautifier reformats it as follows:

```
vector ((1 to 3, 8, 9), 6, databyte:D, byteaddr:H, iocode:O, iocontrol:B,
(43 to 58):H, 59, 60, 61, 62, 63)

 {
 /*@@                                  data  byte  ioc   iocon    44444> 5 6 6 6 6
                            12389   6    byte  addr  ode   trol     34567> 9 0 1 2 3
*************************************************************************************/
0   GLOBAL p1:TSET 1        H1X01   L   .r255 .d1E00.d0   .d0001    .r1234 H 0 1 L H;
1   loop1:  LOOP 10         L1X-1   0   .r253 .d1E02.d755 .d0001    .r5678 0 1 L H 0;
2                           L0X-1   1   .r252 .d1E03.d0   .d0001    .dBA98 1 L H 0 1;
3           REPEAT 35       L1X-1   L   .r251 .d1E04.d0   .d0001    .r1234 L H 0 1 L;
4           END_LOOP loop1  H0X-1   H   .r250 .d1E04.d0   .d0001    .r1234 L H 0 1 L;
5           CALL subr1      L1X-1   0   .r249 .d1E06.d0   .d0001    .r5678 0 1 L H 0;
};
```

■