blazeFilter Manual

Blazepoint DUO & TRIO Series Printers

Document:
Revision 6
26th November 2004

Blazepoint Ltd.

Contents

Introduction to Filters 3
Example 1 4
Example 2 5
Example 3 6
Filter Programming 7
Overview of Filter Operation 7
Filter Configuration Directives 7
Variables and Named Constants 8
Integer Variables 8
Named Integer Constants 8
String Variables 9
Array Variables 9
Labels 9
Search Specifications 10
Filter Instructions 11
Comments 11
General Guidance 12
Filter Instructions 13
Move, Clear, Copy & Append 14
Serial Port operations 15
Integer Arithmetic Operations 16
String Operations 18
String Search Operations 19
Integer / String Conversion Operations 20
Direct Input Operations 21
System Instructions 22
File Search Instructions 23
Program Control Instructions 24
Filter Control Instructions 25
Debugging Instructions 26
Filter Instruction Reference 27
Move/Copy/Append Operations 27
Serial Port Operations 28
Integer Arithmetic Operations 29
Integer Arithmetic with Constant Operand 30
String Operations 31
Integer / String Conversion Operations 32
Direct Input Operations 32
System Instructions 33
File Search Instructions 33
Program Control Instructions 34
Filter Control Instructions 35
Debugging Instructions 35

Introduction to Filters

A filter is essentially a program which runs on the printer reading the incoming data stream and converting all or part of it into a different format. There are several common instances where this facility might prove useful:

1. The printer is being integrated into existing installation which was originally designed for older printers. The customer does not wish to change the system software, or the same software may be required to support a mix of old and new products. If the system software contains commands which are specific to a particular printer type, are no longer supported, or are resolution-dependent, then a filter can be used to modify, replace or rescale some commands.
2. The printer is being integrated into an existing installation which was designed for a different type of printer: perhaps from a different manufacturer, or a different technology altogether such as a dot-matrix line printer. In this case, the data being sent will make no sense to the BPL interpreter, and the filter is used to identify the commands, extract the data and reformat the commands into BPL.
3. The printer requires a simple interface to a peripheral such as a scanner or keyboard, or to an automation system controlled by a PLC. A scanner which sends a single line of data can thus be made to print a label whenever the scanned data is received.

The filter has two components. The search engine scans the incoming data stream for particular patterns. The search specification can be quite complex, looking for text, numbers and control characters. The search engine looks for many patterns simultaneously, and when it finds a match, it starts the filter command processor. This executes the filter instructions for that pattern, which can reformat the matching data, and perhaps gather more data. When the instructions are complete, the search engine starts again.

The filter has two modes of operation while the search engine is running. In *Pass* mode, all data which does not match is passed on to the printer. This would be appropriate where the input consists of BPL commands to which the filter making minor changes. In *Block* mode, nothing is passed through to the printer except what is explicitly sent by the filter command processor. This would be appropriate where the raw input does not contain BPL commands, and must be completely reformatted by the command processor.

Filter definitions are written using a plain text editor and then compiled into the binary form using the blazeConfig application which runs under Windows. The filter programming language, although similar to assembler, is quite easy to use, and filter programs are generally straightforward to write.

Structure of Filter Spec
The filter source file can be considered as a set of search patterns, together with a set of instructions which are executed when the pattern is found in the incoming data stream.

A detailed description is given in the Filter Programming section, but a few simple examples illustrate how the filter might be used in practice. These examples are of necessity very limited in scope, but it is possible to program quite complex filters, and even to emulate a different command language interpreter. Although a fully compatible emulation is unlikely to be achieved, in many applications only a limited subset of the full command language is ever used, so filtering becomes a workable solution.

Example 1

This simple example takes input intended for an older printer type, picks out the obsolete forms of cut and speed command and substitutes the newer forms.

```
Description = "GPL2 Old Command Filter"
FilterMode  = Pass

Int    Speed                    ; Integer register used to hold the print speed
String CutCmd   = "\e*1C"       ; String value holds new CutEveryLabel command

; These first instructions are executed when the filter starts up.
FilterStart:
        Return                  ; Start searching

; Look for an old-style cut command <Esc>C0000 and simply replace it with
; the newer version.

SearchSpec = "\eC0000"
        Copy            CutCmd, Output
        Return

; Look for an old-style speed command in the form <Esc>Snn where
; nn is the speed in 2-digit format. In this command, speed values
; less than 40 were interpreted as (100+value) so to set a speed of
; 125 mm/s the old command was <Esc>S25.

SearchSpec = "\eS%2d"
        ; Read 2 digit speed value starting after 2nd char
        Move            2, %R0
        StrToInt        Match, Dec, 2, Speed
        ; If speed is < 40, add 100 to it.
        Cmp             40, Speed
        BranchIf        GE, NoAdjust
        Add             100, Speed
NoAdjust:
        ; Output newer 3 digit speed command.
        Copy            '\em', Output           ; <Esc> m
        IntToStr        Speed, Dec, 3, Output   ; 3 decimal digits
        Return

; This pattern helps debug by allowing you to turn off the filter
; without having to reboot the printer in safe mode.
SearchSpec = "#FilterOff" Disable
        Filter          Off
        Return
```

Example 2

This is example is used in a service department to ensure that when a product is re-boxed after service, the box shows the same serial number as the product inside. The printer is connected to a barcode scanner on its serial port which scans the product barcode. The filter then generates a simple outer-box label using the scanned data.

```
Description = "Scanner example"
FilterMode  = Block
FilterDebug = 0

; This command is sent to the printer at filter start-up.
; Units=mm, fonts base-aligned, rotation 1, speed 125, Code39 mag 2 3:1, HRI on
String InitCommand = "\eZM\eZB\eV1\em125\eN6231\x02"

; Start of PlaceCode39 command (X = 20, Y = 10, Type 6 = Code39, Height 20 mm)
String PrintCmd1 = "\eB00200010620"

; Terminate Code39, select font 0, point size 16, place text and formfeed
String PrintCmd2 = "q\eY0001601600\eT00200040Serial No.\x04\f"

; These first instructions are executed once when the filter starts up.
FilterStart:
        Copy            InitCommand, Output
        Return          ; Start searching

; Look for any text at the start of a line.
SearchSpec = "%t" LineBegin
        Copy            PrintCmd1, Output           ; Send initial commands
        Copy            Match, Output               ; Send scanned data
        Copy            PrintCmd2, Output           ; Send final commands
        Return

; This pattern helps debug by allowing you to turn off the filter
; without having to reboot the printer in safe mode.
SearchSpec = "#FilterOff" Disable
        Filter          Off
        Return
```

Example 3

Another entirely different application for filters is to allow plain text input from a simple automation system to be merged with a pre-defined label format downloaded using the demo label facility. The advantage of doing this is that the label design does not have to be pre-programmed into the automation system, and can be changed at a later date. The key is to ensure that the elements of raw data sent by the automation system are easily identified, and that the downloaded capture label also has identifiable fields. In this example the raw data is sent one field per line, starting with a colon. The downloaded label has text data fields identified with the string $TEXT-0$, $TEXT-1$ etc. The filter receives the raw data and stores the fields received in numerical order (field 0, field 1, etc), then replays the stored label design substituting the stored fields whenever it encounters the $TEXT-n$ marker.

```
Description = "Data merge"
FilterMode  = Block          ; Don't pass anything yet
FilterDebug = 1              ; Just show sign-on message to aid testing

Const MaxVars = 10           ; Allow for single digit identifiers 0-9
StringArray Vars[MaxVars]    ; Array used to store data from the automation system
Int Index = 0                ; Used to index into the array
Int Indirect                 ; Used for indirect access into the array

FilterStart:
        Return               ; No printer init required, so start searching

; This takes lines of raw input and stores them into 'Vars' in the order received.
; Data starts at the beginning of the line, and ends when any non-text character
; is received.
SearchSpec = ":%t" LineBegin Group = 1
        Cmp             MaxVars, Index          ; Check index is within array bounds
        ReturnIf        GE                      ; Give up if not
        DeleteStr       0, 1, Match             ; Delete the leading ':'
        Move            &Vars, Indirect         ; Get offset of base of array
        Add             Index, Indirect         ; Add the index to get the address
        Copy            Match, [Indirect]       ; Store the data
        Add             1, Index                ; Increment index for next time
        Return

; This term looks for the 'print' command in the raw input.
SearchSpec = "\f" Group = 1
        FileOpen        Capture, 1              ; Open demo file number 1
        Stream          File                    ; Use the file for input
        Search          Disable, 1              ; Disable search group 1
        Search          Enable, 2               ; Enable search group 2
        Filter          Pass                    ; Pass everything from demo label
        Return

; This looks in the demo file for our predefined text string
SearchSpec = "$TEXT-%1d$" Group = 2
        Move            6, %R0                  ; Where to start conversion
        StrToInt        Match, Ref, 1, Index    ; Extract index
        Add             &Vars, Index            ; Convert index to address
        Copy            [Index], Output         ; Send stored text instead
        Return

; At the end of the demo file, we need to revert to checking raw input
FilterEOF:
        Stream          Input                   ; Switch back to raw input
        Search          Disable, 2              ; Disable group 2
        Search          Enable, 1               ; Enable group 1
        Filter          Block                   ; Pass nothing
        Move            0, Index                ; Reset the index for next time
        Return

; This item enables the filter to be disabled without rebooting
SearchSpec = "#FilterOff" Group = 1 Disable
        Filter          Off
        Return
```

Filter Programming

Overview of Filter Operation

The filter processor is an interpreter built into the printer firmware. It behaves like a specialised RISC processor with instruction set optimised for handling the type of string conversions normally used in printer language interpretation.

The filter source file consists of several sections.

Filter directives give the general attributes of the filter such as its title, pass/block mode and debug level. These are normally placed at the top of the filter file.

The filter command processor has 256 integer registers and 256 string registers available. Variable definitions are used to assign names and initial values to any of the filter's registers. These can occur anywhere in the file, but are normally placed at the top of the filter file for easy reference. You can also define integer constants here.

The remainder of the file consists of search specifications and filter processor commands. When the filter starts, the command processor begins executing instructions immediately, starting with the first instruction in the file. It continues until it comes to a Return instruction, whereupon it stops executing, and starts the search engine. When the search engine finds a matching pattern it starts the command processor at the instruction located immediately after the search specification that matched. Again, the command processor continues to execute instructions until it comes to a Return.

For filter debugging purposes, a DebugLevel can be selected which sends information to the serial port during filter execution. This can be displayed on a simple dumb terminal, or a terminal emulation program such as HyperTerminal or TeraTerm.

Filter Configuration Directives

The first few lines of the filter source file contain directives which provide the general configuration.

Description = "*Description of the filter***"**
This description appears as part of the printer self-test report when invoked at power-on, and also in the sign-on message when the debug level is set to any value other than zero. The compiler will also add the name of the source file, and its own version information after the description provided. It is often useful to put your own filter file version number here so that you can check that a printer is loaded with the latest version of your filter file.

StackSize = *Size*
The default stack size of 32 is almost always adequate, and values less than 32 will be ignored anyway. One level of stack is required for each *Call* instruction, so the default provides for up to 31 nested calls. If further nesting is required, eg. For recursive calls (not recommended unless the recursion is carefully controlled) the stack depth can be increased up to a maximum level of 128.

FilterMode = Pass *or* **Block**
This specifies whether the filter will pass data which does not match the search strings while searching is in progress. The default is to pass everything. For non-BPL input, the setting should generally be Block as the data will not make sense to the printer. Filter mode can also be changed at runtime using the *Filter* instruction.

FilterDebug = Level [**Wait**]
This sets the initial debug level. Debug level can also be changed at runtime using the *DebugLevel* instruction, but the filter startup message will only be shown if FilterDebug configuration is non-zero. For definitions of the debug levels, refer to the DebugLevel instruction summary. In general,

debug information should be turned off for a working filter as it slows down the throughput. However, it can be useful at the filter design and testing stage.

The optional keyword *Wait* may be added after the debug level (without square brackets). This controls what happens if debug information is generated faster than it can be output to the serial port. By default, the excess information is discarded, but if the Wait keyword is used, the filter will pause until it can send the information. If a high debug level is set, this is esential to avoid loss of information, but it should not be used in a production filter. If there is no terminal waiting to collect the data, the filter will stall as soon as the internal buffer is full.

Variables and Named Constants

To make programmign easier, the compiler provides for the use of named variables and constants, as with most programming languages. Names may contain letters, numbers and the underscore character, but they may not start with a number.

Integer Variables

The filter processor has up to 256 integer registers. These may be referenced directly by name in an instruction using the form %R*n*. However, to make programming easier, it is strongly recommended that named variables are used instead. The compiler will then assign names to registers automatically. An initial value can also be assigned; if none is given the default is zero.

Examples:

%R4 = 10	Assign initial value 10 to integer register 4.
Int XOffset	Find the first free integer register and give it the name XOffset, with initial value 0.
Int YOffset = 120	Find the first free integer register and give it the name YOffset, with initial value 120.
%R0 Start	Give the name Start specifically to integer register 0. Although it is not normally necessary to specify which register is used, some instructions use %R0 and %R1 for particular purposes, and this form can be used to give a name to these special registers.

Note also that the full range of 256 registers is not available by default. Registers available range from 0 to the highest register index declared in the source file. Instructions which attempt to use out-of-range register indices will simply be ignored at runtime.

All integer variables are 32 bit signed quantities, ie. values lie in the range -2147483648 to 2147483647.

Integer values may be expressed in decimal (optionally signed), or as hexadecimal when preceded by *0x* (zero-x), or as 1 or 2 single-quoted characters. 'A' is equivalent to 0x41, 'AB' is equivalent to 0x4142.

The index of a register may also be used as a constant and is obtained by prefixing the register name with an & sign. eg. if MyReg is %S5, then &MyReg has the constant value 5.

Named Integer Constants

For convenience, integer constants can be given names to make a filter file easier to read. If you use the same value several times then it makes sense to use a named constant instead so the value is only defined once. All constant declarations take the form:
Const Identifier = Value, eg.
Const BoxHeight = 200

As with integer variables, constant values can be expressed as signed decimal, hexadecimal or single-quoted characters.

Note that instructions which use a constant as one of their operands will only accept 8 or 16 bit values, depending on the instruction.

String Variables

String variables are implemented as variable-length text strings, as used in C++, Delphi, Visual Basic, etc. The initial value for a string may be given as a double-quoted string. Uninitialised strings are empty. Non-ascii characters may be represented in the string as 2 hex digits with the prefix \x (backslash-X). A few other characters have special representations as follows :-

Name	Representation	Hex value
Newline / linefeed	\n	<0A>
Formfeed	\f	<0C>
Carriage return	\r	<0D>
Escape	\e	<1B>
Double-quote	\"	<22>
Backslash	\\	<5C>

The following string names have special meaning and are reserved.

Match is the name given to the string where input data matching a successful search is stored. Regardless of filter mode (Pass or Block) data matching the search pattern is never sent to the printer automatically. *Match* is located at string index 0.

Output is the name given to the string used to generate filter output. Copying or appending data to this string sends data directly to the printer. Note that this data is sent and the string cleared immediately after every instruction, so unlike all other strings, there is no logical difference between copy and append for operations on this string. For the same reason, you cannot read anything back from this string. *Output* is located at string index 1.

Integer variables **%R0** and **%R1** are used in many of the string instructions to indicate start position and length. These variables do not have pre-assigned names as their use varies between instructions. You can assign your own names to them if you wish; however, you are advised not to use these registers as general purpose registers.

Array Variables

An array of integers or strings may be created using the syntax:
IntArray Identifier[Size]
StringArray Identifier[Size]
eg. StringArray MyArray[20] creates an array of 20 strings in consecutive locations.

Array variables cannot be initialised. All integers are zero, and strings are empty.

Array variables are useful when storing and retrieving values which you want to access by index number. Refer to example 3 for an application which uses array variables.

Labels

Labels are used as the target for subroutine calls and branches. Naming conventions are as for variables, except that labels names must end with : (colon).

The special label *FilterEOF:* is similar to a search spec and the filter jumps to that label if it reaches the end of an input file while in search mode.

Search Specifications

These begin with the keyword SearchSpec. The pattern to be matched is contained within double quotes, and must be matched exactly, character-for-character, case-sensitively. Within the quotes the \ (backslash) character introduces special sequences exactly as in string variables. In addition, the % (percent) character introduces various wildcard matches as shown in the table below.

Escape sequence	Description
%d	Matches decimal digits. If the d is preceded by a positive number, the search must match exactly that number of digits. If no number is given, this matches one or more decimal digits. If the d is preceded by 0 (zero), this matches zero or more decimal digits.
%x	As above, but matches hexadecimal digits.
%t	As above, but matches any text character, ie. Not a control char.
%?	Matches any single char. If preceded by a positive number, match exactly that number of chars.
%*x	Matches one or more of any character up to and including the first occurrence of *x*. eg. <%*> would match a pair of angle brackets with anything in between. If the * is preceded by 0 (zero) then the condition changes to zero or more of any character except the terminating character.
%%	Matches a literal % sign.

Examples

```
; Match "<Esc>T" followed by exactly 8 digits
SearchSpec = "\eT%8d"

; Match "4911030" at the beginning of a line.
; This spec belongs to group 2.
SearchSpec = "4911030" LineBegin Group = 2

; Match "<Esc>*<Number>X" where <Number> represents 1 or more digits
; Will match <Esc>*21X, <Esc>*1X, <Esc>*1234567X but not <Esc>*X
SearchSpec = "\e*%dX

; Match "<Esc>*<Number>X" where <Number> represents 0 or more digits
; As above, but will also match <Esc>*X
SearchSpec = "\e*%0dX

; Match <Esc>p followed by any single unspecified character
SearchSpec = "\ep%?"

; Match <Esc>p followed by 4 unspecified characters
SearchSpec = "\ep%4?"

; Match one or more characters enclosed in angle brackets
; Will match <ABC>, <1234>, <C5d$> but not <>
SearchSpec = "<%*>"

; Match zero or more characters enclosed in angle brackets
; As above but will also match <>
SearchSpec = "<%0*>"
```

Additional search qualifiers may be added after the search pattern.

LineBegin means that the search can only take place at the beginning of a line. This condition is true at filter startup and after a CR, LF or FF character has been received.

Group = Number Assigns this search spec to a group.
By default all search specs are assigned to group 0. Groups are used by the *Search* instruction to enable or disable a whole group of search terms at once.

Disable means that this search spec is used to disable the filter interpreter. This keyword has no effect on the filter itself, but is used by the filter compiler program, so that it can identify the sequence to send in order to disable the filter for updates.

The search spec must be on a single line, by itself, but line length is essentially unlimited.

When the search engine finds a match between the input data and one of the search specs, the input data which matched the search spec is copied to the *Match* variable. The filter processor then jumps to the instruction immediately after the search pattern specification and continues executing from there. Execution of instructions continues until a return is encountered at the stack base level. If a Call instruction is encountered, then the next Return instruction will jump back to the instruction immediately following the Call instruction. A Return which has no corresponding Call will cause the filter processor to terminate and the search engine to restart.

Labels and SearchSpecs are similar in that they both form jump targets. However, they do not in themselves indicate the end of the previous sequence of instructions. For example:

```
SearchSpec = "CutCommand"
        Copy        BPLCutCommand, Output

SearchSpec = "StatusCommand"
        Copy        BPLStatCommand, Output
        Return
```

If the sequence "CutCommand" is found in the input data, the filter processor will start at the instruction immediately following the first search spec. However, because there is no Return instruction, execution continues with the next instruction, and the filter sends 2 commands instead of the 1 which might be expected.

This feature can be useful as it is possible to associate several search specs with the same set of instructions.

When the filter is in Pass mode, filter searching is disabled temporarily while the printer reads in graphic data. This avoids false positive search triggers when graphics data happens to look like a search pattern. Searching is not disabled for text and barcode content, so it is still possible to get false matching on content. However, this is much less likely to cause problems, and it is easier to guard against it if needed.

Filter Instructions

The remainder of the filter is comprised of sequences of instructions which the filter processor will execute when a search term matches the input data. These do the bulk of the work and are detailed in full in the *Filter Instructions* section.

Comments

A ; (semi-colon) character marks the start of a comment which continues to the end of the line. Use comments liberally to describe what your filter is doing, and why.

General Guidance

Because filtering can impose a significant processing overhead, data throughput may be reduced. However, there are some general guidelines which will help minimise the overhead.

Keep the number of search terms to a minimum. Every byte coming in has to be tested against every search term (when active). A filter with 50 search terms will be slower than one with 5.

Test the *Match* string to differentiate between similar commands rather than specifying separate search terms for each. The search string has to be tested against all incoming data, whereas testing the *Match* string is only done once a search is successful.

For best efficiency, avoid starting search terms with wild-card characters. The search process is optimised to abort quickly if the first character of the search term does not match the incoming data. Using a wild card as the first character will bypass this optimisation.

Do not enable debug output for a finished filter. Debug can generate a large number of output characters for every character processed.

While developing the filter, include a search term which will enable you to switch the filter off by sending a fixed text string to it. The sample filters use the term "#FilterOff" for this purpose, and you are recommended to keep to this standard if possible. The filter compiler includes a facility to send any string to the printer. Generally, an active filter will change the data and make it impossible to download a new version of your filter unless you disable it by command, or power the printer on with the Feed and Pause buttons pressed. When the filter is finished, this search term can be commented out to avoid wasting processing time, although it is best to leave it enabled if possible.

While developing the filter, set the FilterDebug to 1 or more, so that you can see the sign-on message when the filter loads. Include a version number in the filter description, and increment it every time you change the filter. You can use this to verify that a modified version of the filter has been downloaded successfully.

Know what the incoming data looks like. If there is no other way to check it you can power on the printer with the Pause button pressed to enter hexdump mode. This will print the raw unfiltered data instead of interpreting it.

Use comments to document how the filter works so you (or someone else) can modify it at a later date when the requirements change.

Keep the filter simple.

Filter Instructions

This section details all available filter instructions and is arranged in groups of similar instructions. Each instruction starts with the mnemonic followed by its operands, and is shown in a highlighted box similar to the one below.

Move	**Rs, Rd**

The following general notes apply to all instructions.

Naming Conventions
Rx, Ry, Rs and Rd are used to indicate integer registers. These may be explicit registers (%R4), or named registers (XOffset).

Sx, Sy, Ss, and Sd are used to indicate string registers. These may be explicit registers (%S3), or named registers (FormName).

Const16 is used to indicate a 16 bit constant value. The instruction details indicate whether this is interpreted as signed or unsigned.

Source and Destination Operands
The operand on the right is the destination operand, following the Motorola/Hitachi/Zilog assembler conventions. In general this is more intuitive than the Intel convention. Thus Move Rs, Rd means move the contents of Rs to Rd. However, following the same convention, in subtract, compare, divide and remainder operations, the destination is acted on by the source, so Sub A,B performs the operation B – A rather than A – B, which is slightly less intuitive. However, it was felt that mixing two different assembler conventions would only lead to confusion.

Addressing Modes
Most instructions support only direct addressing of registers. However, a few also support indirect addressing, whereby an integer register contains the index of the source/destination register to be used. Indirect addressing is indicated by placing the index register in square brackets.
Example:
R3 contains 5, R4 contains 28, R5 contains 135

Move %R3, %R4	Sets the value of R4 to 5
Move [%R3], %R4	Sets the value of R4 to 135

Condition Codes
The condition code register holds 3 flags which indicate the result of the previous operation. It is these flags which are tested by conditional branch instructions.
Z flag. Set if the result of an operation is 0, or a comparison yields equality.
N flag. Set if the result of an operation is negative, ie. less than 0.
E flag. Set if end-of-file is reached when reading from a file, cleared when input stream changed to standard input or a new file is opened.

Copy and Append
Many string instructions have a Copy and Append variant. The Copy version will replace any existing content in the destination string, whereas the Append version will append to the end of any existing content.

Move, Clear, Copy & Append

All Move, Clear, Copy and Append instructions support indirect addressing.
Move is used for integer variables and Clear, Copy and Append with string variables.

Move	**Rs, Rd**

Copy Rs to Rd, setting the Z and N flags according to the properties of the destination register.

Move	**Const16, Rd**

Move 16 bit immediate data to the destination register, sign-extending to 32 bits. The Z and N flags are set according to the properties of the destination register.

Clear	**Sd**

Clear the content of the destination string register (empty string). No flags are affected.

Copy	**Ss, Sd**

Copy string Ss to Sd, replacing any existing content. No flags are affected.

Append	**Ss, Sd**

Append string Ss to the end of Sd. No flags are affected.

Copy	**Const16, Sd**
Append	**Const16, Sd**

Copy/append 1 or 2 characters represented by Const16 to Sd. No flags are affected.

Examples:

```
Int x = 3
Int y
String Name   = "ABC"
String Serial = "123"
String Desc   = "XYZ"
Move            x, y                    ; y = 3
Move            23, y                   ; y = 23
Clear           Desc                    ; Desc is empty
Copy            Name, Desc              ; Desc = "ABC"
Append          ':-', Desc              ; Desc = "ABC:-"
Append          Serial, Desc            ; Desc = "ABC:-123"
Move            &Serial, x              ; x contains register number of Serial
Copy            [x], Desc               ; Desc = "123"
```

Serial Port operations

These instructions permit data to be written to and read back directly from the serial port. This can be used if there is a scanner or other peripheral device attached to the port.

SerialOut	**Ss**

Copy source string Ss to the serial port. No flags are affected.

SerialOut	**Const16**

Copy 1 or 2 characters represented by Const16 to the serial port. No flags are affected.

SerialLock	**Mode**

Lock/unlock the serial port input to prevent the auto input selector from taking any data from it. The port must be locked before using the SerialIn/SerialAppend instructions. No flags are affected. Mode can be "Disable" (= 0) or "Enable" (=1).

SerialIn	**Sd**
SerialAppend	**Sd**

Copy/append data from the serial port input buffer to the destination string. No flags are affected. This instruction should only be used when the main data input comes from another source. Typically used when a scanner is attached to the serial port for barcode verification.

Examples:

```
String ReadBack
String Desc    = "XYZ"
SerialLock      On                      ; Lock port for input
SerialOut       Desc                    ; Send "XYZ" to serial port
SerialOut       '\r\n'                  ; Send CR-LF to serial port
SerialIn        ReadBack                ; Get data from port
SerialLock      Off                     ; Port reverts to normal use
```

Integer Arithmetic Operations

In all arithmetic operations, the Z and N flags are set according to the result of the operation.

16 bit immediate operands are always sign extended to 32 bits before performing the operation.

Add **Rx, Rd**

Rd = Rd + Rx.

Sub **Rx, Rd**

Rd = Rd - Rx.

Cmp **Rx, Rd**

Flags are set as if the operation Rd - Rx had been performed, but the destination register is unchanged.

Mul **Rx, Rd**

Rd = Rd * Rx.

Div **Rx, Rd**

Rd = Rd / Rx, ie. the integer result, discarding the remainder. If the divisor is zero, the result is saturated to the largest positive or negative integer (depending on signs of operands).

Rem **Rx, Rd**

Rd = Rd % Rx, ie. the remainder after performing the division. If the divisor is zero, the result is always zero.

Max **Rx, Rd**

Rd = the larger of Rd & Rx.

Min **Rx, Rd**

Rd = the smaller of Rd & Rx.

Add **Const16, Rd**

Rd = Rd + Const16.

Sub **Const16, Rd**

Rd = Rd - Const16.

SubR **Const16, Rd**

Rd = Const16 - Rd.

Neg **Rd**

Rd = - Rd.

Cmp **Const16, Rd**

Flags are set as if the operation Rd – Const16 had been performed, but the destination register is unchanged.

Mul **Const16, Rd**

Rd = Rd * Const16.

Div	Const16, Rd

Rd = Rd / Const16, ie. the integer result, discarding the remainder.

Rem	Const16, Rd

Rd = Rd % Const16, ie. the remainder after performing the division.

Floor	Const16, Rd

Rd = the larger of Rd & Const16. Effectively this limits Rd to being no lower than a constant floor.

Ceil	Rx, Ry, Rd

Rd = the smaller of Rd & Const16. Effectively this limits Rd to being no higher than a constant ceiling.

Examples:

```
Int x = 3
Int y = 7
Int z = 32
Add             x, y                    ; y = 10 (3 + 7)
Sub             y, z                    ; z = 22 (32 - 10)
Rem             x, z                    ; z = 1 (remainder of 22 / 3)
Add             12, x                   ; x = 15 (12 + 3)
Max             x, y                    ; y = 15 (max of 10 and 15)
Sub             16, y                   ; y = -1 (15 - 16)
SubR            16, x                   ; x = 1 (16 - 15)
Floor           0, y                    ; y = 0 (max of 0 and -1)
Ceil            27, x                   ; x = 1 (min of 1 and 27)
```

String Operations

SubStr	Ss, Offset, Length, Sd

Copy Length characters from source string Ss into destination string Sd, starting at the position indicated by Offset. Offset can be either 0 or the value in R0. Length can be an 8-bit unsigned number (0 to 255) or the value in R1. If Length is zero, the copy continues to the end of the source string. After the copy, R0 is updated to index the character **after** the last one copied, and R1 contains the number of characters copied. No flags are affected.

SubStrAppend	Ss, Offset, Length, Sd

Identical to SubStr except that the resulting sub-string is appended to the existing content of Sd rather than replacing it.

DeleteStr	Offset, Length, Sd

Delete characters from destination string Sd, starting at the position indicated by Offset. Offset can be either an 8-bit unsigned number (0 to 255) or an 8-bit signed number relative to the value in R0. Length can be an 8-bit unsigned number (0 to 255) or the value in R1. If the resultant length is zero, all characters are deleted from Offset to the end of the string. After deletion, R1 contains the number of characters actually deleted. R0 is unchanged, and no flags are affected.

```
Examples:
String Descr = "steel M4 lock nut"
String Temp
SubStr          Descr, 6, 2, Temp       ; Temp = "M4", %R0 = 8, %R1 = 2
SubStrAppend    Descr, %R0+5, 0, Temp   ; Temp = "M4 nut", %R0 = 17, %R1 = 4
DeleteStr       5, %R1, Descr           ; Descr = "steellock nut", %R1 = 4
```

CmpStr	Sx, Sy

Performs a direct case-sensitive comparison of the two strings and sets the Z flag accordingly. Both Sx and Sy can use direct or indirect addressing.

CmpStrNoCase	Sx, Sy

Identical to CmpStr, except that the compare is not sensitive to case.

StrLen	Ss, Offset, Rd

Calculate the length of Ss, starting from the given offset, and store the result in Rd. Normally Offset will be zero, but it is sometimes useful to count the remaining length of the string relative to a fixed offset, or relative to the current position indexed by %R0. %R0 and %R1 are not changed by this instruction. The Z flag is set if length is zero, otherwise it is cleared.

Examples:

```
String Str1 = "Example"
String Str2 = "example"
Int Len
CmpStr          Str1, Str2              ; Z flag cleared - no match
CmpStrNoCase    Str1, Str2              ; Z flag set - strings match
StrLen          Str1, 2, Len            ; Len = 5 (Length of "ample")
```

String Search Operations

The Offset parameter indicates the starting position in the string. The first character is considered to be at offset 0. The offset can be specified as an absolute value in the range 0 to 255, or it can be specifed relative to the value in %R0, with an offset in the range –128 to 127. This value is added to %R0 at the start of the operation.

FindChr **Ch, Sy, Offset**

Search string Sy from the position indicated by Offset for the first occurrence of character Ch. The offset can be specified directly or relative to %R0. If the character is found, the Z flag is set and %R0 is set to the index in Sy where the character was found. Otherwise the Z flag is cleared.

FindStr **Sx, Sy, Offset**

Search string Sy from the position indicated by Offset for the first occurrence of sub-string Sx. The offset can be specified directly or relative to %R0. If the sub-string is found, the Z flag is set and %R0 is set to the index in Sy where the character was found. Otherwise the Z flag is cleared.

FindMember **Sx, Sy, Offset**

Search string Sy from the position indicated by Offset for the first occurrence of any character which is a member of string Sx. The offset can be specified directly or relative to %R0. If a character is found, the Z flag is set, %R0 is set to the index in Sy where the character was found and %R1 is set to the index in Sx of the member which was found. Otherwise the Z flag is cleared.

FindNonMember **Sx, Sy, Offset**

Identical to FindMember, except that the search is for the first character in Sy which is not a member of Sx. %R1 is not changed by this instruction.

Examples:

```
String Descr = "steel M4 lock nut stock"
String Num = "0123456789"
String Test = "ock"
String MemTest = "M4le stock"
FindChr         'l', Descr, 0           ; %R0 = 4, Z flag set
FindChr         'l', Descr, %R0         ; %R0 = 9, Z flag set
FindStr         Test, Descr, %R0        ; %R0 = 10, Z flag set
FindStr         Test, Descr, %R0        ; %R0 = 10, Z flag set
FindStr         Test, Descr, %R0+1      ; %R0 = 20, Z flag set
FindStr         Test, Descr, %R0+1      ; Z flag cleared
FindMember      Num, Descr, 0           ; %R0 = 7, %R1 = 4, Z flag set
FindNonMember   MemTest, Descr, 0       ; %R0 = 14, Z flag set
```

Integer / String Conversion Operations

StrToInt Ss, Type, Length, Rd

Convert Length characters from source string Ss into an integer value in destination register Rd, starting at the offset value in R0. Length can be a 4-bit unsigned number (0 to 15) or the value in R1. If Length is zero, the conversion continues to the end of the source string or until a non-convertible character is found. After the conversion, R0 is updated to index the character **after** the last one used, and R1 contains the number of characters used. The N and Z flags are updated to reflect value in the destination register.

Type	Conversion
Dec	Decimal (digits 0-9) , base 10
Hex	Hexadecimal (0-9, A-F or a-f), base 16
Bin	Binary (digits 0-1), base 2
Ref	Reference ID (0-9, A-Z, a-z) base 62. Generally used for a single character.
RawBE or Ascii	Raw data, big-endian, ie. most significant byte (big end) first. Ascii is as synonym used to extract the ascii value of a character.
RawLE	Raw data, little endian, ie. least significant byte (little end) first.

There is no way to specify the start position within the instruction itself. Remember to set R0 first to mark the position you want to start the conversion from.

Examples:

```
String Data = "1012F\x12\x34"
Int Result
Move            0, %R0                  ; Start conversion at beginning of string
StrToInt        Data, Dec, 2, Result    ; Result = 10, %R0 = 2, %R1 = 2
StrToInt        Data, Dec, 3, Result    ; Result = 12, %R0 = 4, %R1 = 2
Move            0, %R0                  ; Start again ...
StrToInt        Data, Bin, 0, Result    ; Result = 5, %R0 = 3, %R1 = 3
StrToInt        Data, Hex, 20, Result   ; Result = 47 (0x2F), %R0 = 5, %R1 = 2
StrToInt        Data, RawBE, 0, Result  ; Result = 0x1234, %R0 = 7, %R1 = 2
Move            5, %R0                  ; Same place ...
StrToInt        Data, RawLE, 0, Result  ; Result = 0x3412, %R0 = 7, %R1 = 2
Move            0, %R0                  ; Start again ...
StrToInt        Data, Ascii, 0, Result  ; Result = 0x31 (ascii value of '1')
```

IntToStr Rs, Type, Length, Sd

This is the inverse operation to StrToInt. Convert the register to a character array, using leading zero padding if necessary to fill at least Length characters. Length can be a 4-bit unsigned number (0 to 15) or the value in R1. If Length is zero, the conversion produces at least one character. After the conversion R1 contains the number of characters output. No flags are affected.

IntToStrAppend Rs, Type, Length, Sd

This is identical to IntToStr except that output is appended instead of replacing existing content.

Examples:

```
Int Value = 125
String Result
IntToStr        Value, Hex, 0, Result   ; Result = "7D", %R1 = 2
IntToStrAppend  Value, Dec, 5, Result   ; Result = "7D00125", %R1 = 5
```

Direct Input Operations

These operations all take characters directly from the input stream rather than from a string. This might be used instead of the event-based pattern matching, or subsequent to finding a match.

All these commands have 3 variants: Copy, Append and Skip, of which only Copy is described in detail. In the Append variant, output is appended to the destination string rather than replacing it. In the Skip variant, input characters are consumed as before, but then discarded rather than being stored to a string variable. In all variants, if input is being read from a file rather than the comms input, the instruction terminates if end-of-file is reached.

CopyIn	**Length, Sd**
AppendIn	**Length, Sd**
SkipIn	**Length**

Copies a fixed number of characters from the input stream. Length can be specified as an unsigned 16 bit number, or the value in R1 can be used.

CopyInUntil	**Ch, Length, Sd**
AppendInUntil	**Ch, Length, Sd**
SkipInUntil	**Ch, Length**

Copies characters from the input stream until a specific character is found, or a maximum number of characters has been read. Length can be specified as an unsigned 8-bit number, or the value in R1 can be used. A length of zero signifies no limit on length. After the operation, R1 reflects the number of characters read. If the search character was found, the Z flag is set. The search character is **not** included in the output and remains in the input stream.

CopyInUntilMember	**Sx, Length, Sd**
AppendInUntilMember	**Sx, Length, Sd**
SkipInUntilMember	**Sx, Length**

Works in the same way as CopyInUntil, but rather than looking for a single character as a stop condition, any character that is a member of the string Sx will end the input.

CopyInWhileMember	**Sx, Length, Sd**
AppendInWhileMember	**Sx, Length, Sd**
SkipInWhileMember	**Sx, Length**

Works in the same way as CopyInUntilMember, but uses the inverse termination condition. Input stops when a character which is not a member of Sx is encountered.

System Instructions

These instructions provide a means to get information from the printer sub-system.

SysStr SourceID, Sd

Copies a system variable to a string variable. This provides a means to read the printer model or serial number. It can be used to restrict a filter to being used on one type of printer, or even on one particular printer. No flags are affected.

SourceID	Meaning
$PrinterModel	Model name, eg. "DUO 2104"
$PrinterOEM	Manufacturer, eg. "Blazepoint"
$OEMkey	Key which can be assigned by OEM
$Firmware	Version number eg. "V1.35"
$PrinterSerial	Unit serial number, eg. "52010103001"

SysInt SourceID, Rd

Copies a system variable to an integer variable. This provides a means of synchronizing with the printer sub-system. The Z and N flags are set according to the properties of the destination register.

SourceID	Meaning
$LabelStatus	Read the print status of the current label
$FormLength	Feed distance of 1 label in units of 0.01 mm
$LabelCounter	Printer lifetime label counter
$LengthCounter	Printer lifetime media feed counter in units of 10 mm
$TransferMode	0 = Direct thermal, 1 = thermal transfer
$DefaultSpeed	Default print speed in mm/s
$DefaultHeat	Default print heat as a percentage (50 – 200)
$HeadWidth	Printhead width in pixels
$DotsPerMM	Printhead resolution in dots (pixels) per mm
$XOffset	Current label XOffset value in pixels
$LabelHeight	Height of the label in pixels (formlength – gap)

LabelStatus reflects the progress of a label through the print engine. Successive reads of LabelStatus will track label progress, and once 'Finished printing' status has been read, it will reset to 'Waiting'. Sensible results can only be obtained when printing one label at a time.

Value	Meaning
0	Waiting. Nothing to do, paper movement stopped
10	Started printing
20	Finished printing. May still be feeding
100	Any error condition

File Search Instructions

This group of file search instructions works with filenames in the form Directory/Number.Name, as used in the capture and keyboard directories. Source can be "Capture" (=0) or "Keyboard" (=1). After a search, the number of matching files is stored in %R0. If files are found, the file number of the first file found is stored in %R1, and the file number & name is stored in Sd. The flags are set as if `Cmp 1,%R0` had been executed, so use `BranchIf LT` to check for no files, `BranchIf EQ` to check for exactly 1 file, and `BranchIf GT` to check for multiple files.

All file name searches are case-insensitive.

FindFileNum **Source, Sy, Sd**

Searches the source directory for the file number extracted from Sy. Sy may contain decimal digits terminated with a dot (eg. "2.Untitled") and then followed by anything, or terminated by the end of string (eg. "2").

FindFileName **Source, Sy, Sd**

Searches the source directory for the file named in Sy, with no preceding number, (eg. "Untitled").

FindFileSubName **Source, Sy, Sd**

Searches the source directory for the files whose names include the substring in Sy. Sy may be empty, in which case any file will match.

FindNextFile **Sd**

After using any of the instructions above to find a file, this instruction may be used to extract other files matching the same spec. If a file is found the Z flag is set, the file number is stored in %R1 and the file number & name is stored in Sd. Otherwise the Z flag is cleared.

Examples:
Assume the printer file system includes the following keyboard files:

Keyboard/1.Dispatch
Keyboard/2.Fragile
Keyboard/3.Serial

```
String Filename
String FNum = "2"
String FName = "SERIAL"
String Blank

FindFileNum     Keyboard, FNum, Filename   ; Filename="2.Fragile", %R0=1, %R1=2
FindFileName    Keyboard, FName, Filename  ; Filename="3.Serial", %R0=1, %R1=3
FindFileSubName Keyboard, Blank, Filename  ; Filename="1.Dispatch", %R0=3, %R1=1
FindNextFile    Filename                   ; Filename="2.Fragile ", %R1=2, Z set
FindNextFile    Filename                   ; Filename="3.Serial", %R1=3, Z set
FindNextFile    Filename                   ; Z clear (no more files)
```

Program Control Instructions

These instructions perform conditional or unconditional branches to a different part of the code. The address must be a valid label. For conditional branches, use on of the condition codes listed below. The branch instruction itself does not affect any flags.

Name	Flags	Description
Z, EQ	Z set	Result is zero, or compare equal
NZ, NE	Z clear	Result is non-zero
GT	Z and N clear	Result > 0, or compare dest > src
GE	N clear	Result >= 0, or compare dest >= src
LT	N set	Result < 0, or compare dest < src
EOF	E flag set	End of file reached
NEOF	E flag clear	End of file not reached

Branch **Label**
BranchIf **Condition, Label**

If the specified condition is met, branch directly to the label specified, otherwise continue with the next instruction as normal. Addressing uses one address per instruction.
Example:

```
BranchIf        NZ, YourLabel           ; Branch if Z flag not set
Branch          MyLabel                 ; Always branch
```

BranchIf **Rs, Value, Label**

This instruction compares Value with the contents of R0 or R1 and branches if they match. This may be used to set up a branch table, either to take different branches based on a simple integer variable, or in conjunction with the FindStr and FindMember commands to perform different actions depending on the character or substring found.
Example:
Assume %R1 contains the value 2

```
BranchIf        %R1, 1, Action1         ; Branch not taken
BranchIf        %R1, 2, Action2         ; Branch taken
BranchIf        %R1, 3, Action3         ; Never get to this instruction
```

Call **Label**
CallIf **Condition, Label**

This operates in the same way as the branch instruction except that program counter is saved on the stack and can be restored using a corresponding Return instruction. This allows sub-routine calls to be implemented.

Return
ReturnIf **Condition**

After a Call, this returns the program counter to the value saved on the stack, provided the specified condition is met. Normally the unconditional variant is used. If the stack is already at base level, ie. there are no more sub-routines to return from, instruction execution stops and the search engine restarts.

Filter Control Instructions

Filter **Mode**

This determines how the search engine operates while not executing program opcodes. No flags are affected.

Name	Mode	Description
Off	0	Event processing is disabled and the filter is effectively offline. This is typically used along with a specific search string eg. “#FilterOff” to allow the filter to be disabled so that a new version can be uploaded without having to reboot the printer in safe mode.
Pass	1	During event searching, all characters rejected by the search are passed through to the printer. Typically used with a filter which has to modify BPL input in some way.
Block	2	During event searching, all characters rejected by the search are discarded. The only input seen by the printer is that explicitly generated by the filter program. Typically used when filtering input intended for another printer type, eg. ZPL, DPL, EPL.

FileOpen **Source, Ident**
FileOpen **Source, Rs**

A capture or keyboard file can be used as an alternate input stream, as a means of merging unformatted data with a label design. This instruction selects a file to be used as the alternate input stream and opens it. Any file already open is closed automatically – there is no separate FileClose command. Ident refers to the file number within the directory, eg. Capture/32.Untitled has ident=32. In the Capture directory, if Ident is zero, the current file selected from the front panel is used. Ident can also be taken from a register, Rs. Source can be "Capture" (=0) or "Keyboard" (=1).

Stream **Source**

This command switches between the two streams. Source can be “Input” (=0) or “File” (=1). The EOF flag is reset by this command. When a file is open you can switch between Input and File any number of times.

Search **State, Group**

This instruction enables or disables a group of Search events. By default, all search events belong to group 0, but can be assigned to any group number 0 to 255 in the SearchSpec statement. State can be "Disable" (= 0) or "Enable" (=1).

For a worked example on how the instructions above are used, see Example 3.

Delay **Msec**
Delay **Rs**

Generates a time delay in msec. Resolution is typically 100 msec, and the period given will be rounded to the nearest unit of resolution. The delay will always be at least one unit of resolution.

FlushIn

Delete any characters remaining in the input buffer. This is sometimes necessary in interactive applications such as keyboard processing.

Debugging Instructions

DebugLevel	**Level**
DebugLevel	**Rs**

This instruction changes the amount of debug information sent to the serial port. A finished filter should have debug level 0 (the default) to disable all output. Producing debug output can seriously slow down a filter, particularly at the higher levels. The level may be specified directly or taken from the value in register Rs.

Level	Description
0	No debug output
1	Startup message and DebugShow messages
2	All events displayed
3	All instructions displayed
4	All output to printer displayed

If a large amount of debug information is required, use the **Wait** parameter in the DebugLevel directive to ensure that no output is lost. Be sure to disable this before releasing the filter, as it will stall if there is no output device connected.

DebugShow	**Register**
DebugShowStr	**Register**

This instruction displays the content of a specific register. Valid for R0-R255, S0-S255, CCR. If CCR is selected, the instruction will also display the program counter, the stack level and stack contents. This output is suppressed at Debug level 0. Indirect addressing is supported. In most instances, DebugShow will select integer or string output automatically depending on whether 'Register' is an integer or string variable. In the case where indirect addressing is used, DebugShow is taken to mean the address of an integer, and DebugShowStr is taken to mean the address of a String.

Filter Instruction Reference

This section simply documents the instruction word structure for each filter instruction.

Every instruction is contained in a 32-bit fixed length instruction word. The 5 MSBs contain the main opcode, the next 3 bits normally hold mode flags or sub-function codes, and the remaining 24 bits hold the operands.

Where flags are present in bits 24-26 the state should be interpreted as follows:
Addressing mode: 0 = direct, 1 = indirect
Append flag: 0 = copy, 1 = append
Skip flag: 0 = copy/append, 1 = skip
Source offset mode: 0 = direct or zero, 1 = relative to %R0
Source length mode: 0 = direct, 1 = value in %R1

Move/Copy/Append Operations

Move Rs, Rd

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 0 0 0 0	0 S D	Rd	0	Rs

S = Source addressing mode, D = Destination addressing mode.

Move Const16, Rd

31 – 27	26 – 24	23 – 16	15 – 0
0 0 0 0 0	1 0 D	Rd	Const16

D = Destination addressing mode.

Clear Sd

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 0 0 0 0	1 1 D	Sd	0	0

D = Destination addressing mode.

Copy Ss, Sd
Append Ss, Sd

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 0 0 0 1	A S D	Sd	0	Ss

A = AppendFlag, S = Source addressing mode, D = Destination addressing mode.

Copy Const16, Sd
Append Const16, Sd

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 0 0 1 0	A 0 D	Sd	Const16	

A = AppendFlag, D = Destination addressing mode.

Serial Port Operations

SerialOut Ss

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 0 0 1 1	0 0 S	Ss	0	0

S = Source addressing mode.

SerialOut Const16

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 0 0 1 1	0 1 0	0	Const16	

SerialLock Mode

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 0 0 1 1	1 0 0	0	Mode	

SerialIn Sd
SerialAppend Sd

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 0 0 1 1	A 1 1	Sd	0	

A = AppendFlag.

Integer Arithmetic Operations

The opcode takes 3 operands, but as this proved not to be very useful in practice, the compiler supports the more common 2 operand version, where the middle operand is replaced by the destination operand, so Add Rx, Rd is synonymous with Add Rx, Rd, Rd

Add Rx, Ry, Rd

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 0 1 0 0	0 0 0	Rd	Ry	Rx

Sub Rx, Ry, Rd

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 0 1 0 0	0 0 1	Rd	Ry	Rx

Cmp Rx, Ry

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 0 1 0 0	0 1 0	0	Ry	Rx

Mul Rx, Ry, Rd

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 0 1 0 0	0 1 1	Rd	Ry	Rx

Div Rx, Ry, Rd

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 0 1 0 0	1 0 0	Rd	Ry	Rx

Rem Rx, Ry, Rd

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 0 1 0 0	1 0 1	Rd	Ry	Rx

Max Rx, Ry, Rd

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 0 1 0 0	1 1 0	Rd	Ry	Rx

Min Rx, Ry, Rd

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 0 1 0 0	1 1 1	Rd	Ry	Rx

Integer Arithmetic with Constant Operand

Add Const16, Rd

31 – 27	26 – 24	23 – 16	15 – 0
0 0 1 1 0	0 0 0	Rd	Const16

Note: Sub Const16,Rd is encoded as Add –Const16,Rd

SubR Const16, Rd

31 – 27	26 – 24	23 – 16	15 – 0
0 0 1 1 0	0 0 1	Rd	Const16

Note: Neg Rd is encode as SubR 0, Rd

Cmp Const16, Rd

31 – 27	26 – 24	23 – 16	15 – 0
0 0 1 1 0	0 1 0	Rd	Const16

Mul Const16, Rd

31 – 27	26 – 24	23 – 16	15 – 0
0 0 1 1 0	0 1 1	Rd	Const16

Div Const16, Rd

31 – 27	26 – 24	23 – 16	15 – 0
0 0 1 1 0	1 0 0	Rd	Const16

Rem Const16, Rd

31 – 27	26 – 24	23 – 16	15 – 0
0 0 1 1 0	1 0 1	Rd	Const16

Floor Const16, Rd

31 – 27	26 – 24	23 – 16	15 – 0
0 0 1 1 0	1 1 0	Rd	Const16

Ceil Const16, Rd

31 – 27	26 – 24	23 – 16	15 – 0
0 0 1 1 0	1 1 1	Rd	Const16

String Operations

CmpStr Sx, Sy

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 1 0 0 0	0 Y X	0	Sy	Sx

Y = Sy addressing mode, X = Sx addressing mode

CmpStrNoCase Sx, Sy

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 1 0 0 0	1 Y X	0	Sy	Sx

Y = Sy addressing mode, X = Sx addressing mode

FindChr Ch, Sy, Offset

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 1 0 0 1	0 0 M	Offset	Sy	Ch

M = Source offset mode

FindStr Sx, Sy, Offset

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 1 0 0 1	0 1 M	Offset	Sy	Sx

M = Source offset mode

FindMember Sx, Sy, Offset

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 1 0 0 1	1 0 M	Offset	Sy	Sx

M = Source offset mode

FindNonMember Sx, Sy, Offset

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 1 0 0 1	1 1 M	Offset	Sy	Sx

M = Source offset mode

StrLen Ss, Offset, Rd

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 1 0 1 0	0 0 M	Rd	Offset	Ss

M = Source offset mode

SubStr Ss, Offset, Length, Sd

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 1 1 0 0	0 L M	Sd	Len	Ss

M = Source offset mode, L = Source length mode

SubStrAppend Ss, Offset, Length, Sd

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 1 1 0 0	1 L M	Sd	Len	Ss

M = Source offset mode, L = Source length mode

DeleteStr Offset, Length, Sd

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
0 1 1 0 1	0 L M	Sd	Len	Offset

M = Source offset mode, L = Length addressing mode

Integer / String Conversion Operations

StrToInt Ss, Type, Len, Rd

31 – 27	26 – 24	23 – 16	15 – 12	11 – 8	7 – 0
0 1 1 1 0	0 0 L	Rd	Type	Len	Ss

L = Source length mode

Type	Mnemonic
0	Dec
1	Hex
2	Bin
3	Ref
4	RawBE or Ascii
5	RawLE

IntToStr Rs, Type, Len, Sd
IntToStrAppend Rs, Type, Len, Sd

31 – 27	26 – 24	23 – 16	15 – 12	11 – 8	7 – 0
0 1 1 1 1	A 0 L	Sd	Type	Len	Rs

A = Append flag, L = Source length mode

Direct Input Operations

CopyIn Length, Sd
AppendIn Length, Sd
SkipIn Length

31 – 27	26 – 24	23 – 16	15 – 0
1 0 0 0 0	A S L	Sd	Length

A = Append mode, S = Skip mode, L = Source length mode

CopyInUntil Ch, Length, Sd
AppendInUntil Ch, Length, Sd
SkipInUntil Ch, Length

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
1 0 0 0 1	A S L	Sd	Length	Ch

A = Append mode, S = Skip mode, L = Source length mode

CopyInUntilMember Sx, Length, Sd
AppendInUntilMember Sx, Length, Sd
SkipInUntilMember Sx, Length

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
1 0 0 1 0	A S L	Sd	Length	Sx

A = Append mode, S = Skip mode, L = Source length mode

CopyInWhileMember Sx, Length, Sd
AppendInWhileMember Sx, Length, Sd
SkipInWhileMember Sx, Length

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
1 0 0 1 1	A S L	Sd	Length	Sx

A = Append mode, S = Skip mode, L = Source length mode

System Instructions

SysStr SourceID, Sd

31 – 27	26 – 24	23 – 16	15 – 0
1 0 1 0 0	0 0 0	Sd	SourceID

SourceID	Value
PrinterModel	0
PrinterOEM	1
OEMkey	2
Firmware	3
PrinterSerial	4

SysInt SourceID, Rd

31 – 27	26 – 24	23 – 16	15 – 0
1 0 1 0 0	0 1 0	Rd	SourceID

SourceID	Value
LabelStatus	0
FormLength	1
LabelCounter	2
LengthCounter	3
TransferMode	4

File Search Instructions

FindFileNum Source, Sy, Sd

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
1 1 0 0 0	0 0 0	Sd	Sy	Source

FindFileName Source, Sy, Sd

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
1 1 0 0 0	0 0 1	Sd	Sy	Source

FindFileSubName Source, Sy, Sd

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
1 1 0 0 0	0 1 0	Sd	Sy	Source

FindNextFile Sd

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
1 1 0 0 0	0 1 1	Sd	0	0

Program Control Instructions

Conditions are checked by applying Mask to the CCR and comparing the result with Cond. Setting both Mask and Cond to zero effectively provides an unconditional branch.
Mask and Cond bits are in this order in the instruction word [0 E N Z]

Branch Address
BranchIf Cond, Address

31 – 27	26 – 24	23 – 20	19 – 16	15 – 0
1 1 1 0 0	0 0 0	Mask	Cond	Address

BranchIf Rs, Value, Address

31 – 27	26 – 24	23 – 16	15 – 0
1 1 1 0 0	0 1 R	Value	Address

R: 0 = R0, 1 = R1

Call Address
CallIf Cond, Address

31 – 27	26 – 24	23 – 20	19 – 16	15 – 0
1 1 1 0 0	1 0 0	Mask	Cond	Address

Return
ReturnIf Cond

31 – 27	26 – 24	23 – 20	19 – 16	15 – 0
1 1 1 0 0	1 0 1	Mask	Cond	0

Filter Control Instructions

Filter Mode

31 – 27	26 – 24	23 – 20	19 – 16	15 – 0
1 1 1 0 1	0 0 0	0 0 0 0	Mode	0

Modes: 0 = Off, 1 = Pass, 2 = Block

Stream Source

31 – 27	26 – 24	23 – 20	19 – 16	15 – 0
1 1 1 0 1	0 0 1	0 0 0 0	Source	0

Source can be “Input” (=0) or “File” (=1).

FileOpen Source, Ident

31 – 27	26 – 24	23 – 16	15 – 0
1 1 1 0 1	0 1 S	Ident or Rs	Source

S = Ident addressing mode: 0 = direct ident, 1 = ident in Rs
Source can be "Capture" (=0) or "Keyboard" (=1).

Search State, Group

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
1 1 1 0 1	1 0 0	Group	0	State

State: 0 = Off, 1 = On

Delay Msec
Delay Rs

31 – 27	26 – 24	23 – 16	15 – 0
1 1 1 0 1	1 0 1	Rs	Msec

If Msec = 0, use value in Rs

FlushIn

31 – 27	26 – 24	23 – 16	15 – 0
1 1 1 0 1	1 1 0	0	0

Debugging Instructions

DebugLevel Level

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
1 1 1 1 1	0 0 M	Level or Rd	0	0

M = Mode: 0 = direct level, 1 = level in Rd

DebugShow Register
DebugShowStr Register

31 – 27	26 – 24	23 – 16	15 – 8	7 – 0
1 1 1 1 1	0 1 S	Rs or Ss	0	RegType

RegType: 0=Int, 1=String, 2=CCR, PC, SP & Stack.
S = Source addressing mode