1. Introduction

1.1 Contents

cont'd.

UBI Fingerprint 7.11
Programmer's Guide
Edition 1, February 1998
Part No. 1-960454-00

1.1 Contents, cont'd.

cont'd.

1.1 Contents, cont'd.

cont'd.

1.1 Contents, cont'd.

Information in this manual is subject to change without prior notice and does not represent a commitment on the part of Intermec Printer AB.

1.2 Preface

UBI Fingerprint 7.11 is a new version of a well-known Basic-inspired, printer-resident programming language that has been developed for use with computer-controlled direct thermal and thermal transfer printers manufactured by United Barcode Industries (UBI). UBI Fingerprint 7.xx works only with the new generation of RISC-processor based printers, starting with EasyCoder 501 XP and EasyCoder 601 XP.

The UBI Fingerprint software is an easy-to-use intelligent programming tool for label formatting and printer customizing, which allows you to design your own label formats and write your own printer application software.

You may easily create a printer program by yourself that exactly fulfils your own unique requirements. Improvements or changes due to new demands can be implemented quickly and without vast expenses.

The new UBI Direct Protocol 7.11 is used for combining variable input data from a host with predefined label layouts.

This tutorial manual describes how to start up UBI Fingerprint programming and how to use the various instructions in their proper context. Programming instructions are explained only briefly. The UBI Direct Protocol 7.11 is described in a separate Programmer's Guide.

The UBI Fingerprint ≥ 7.11 Reference Manual contains detailed information on all programming instructions in the UBI Fingerprint programming language in alphabetical order. It also contains other types of program-related information that are common for all printer models from UBI that uses the corresponding version of UBI Fingerprint.

All information needed by the operator, like how to run the printer, how to load the paper supply and how to maintain the printer, can be found in the User's Guide and Installation & Operation manual for the printer model in question.

The Installation & Operation manual for each printer model also provides information on installation, setup, density, paper specifications, positioning, and other technical data, which are specific for the printer model in question.

UBI Fingerprint 7.11 also supports:

- **UBI Shell 4.1**
  Startup program for EasyCoder 501 XP/601 XP printers
- **UBI LabelShop**
  Various versions
- **UBI Windows Driver**
  For using an EasyCoder printer with most programs run under MS Windows 3.11 and Windows 95.

1.3 News in UBI Fingerprint 7.11

UBI Fingerprint 7.11 is the first publicly documented version of the new generation of UBI Fingerprint developed for use in the EasyCoder XP series.

General changes:
- New CPU board architecture with FLASH memory.
- New printout handling with two large image buffers and no banding.
- Communication port "uart3:" and "rs485:" no longer supported.
- New font handling with scalable fonts and font aliases (see FONT stmt).
- Double-byte fonts support (see FONTD and NASCD stmts).
- Some new devices added, others deleted (see Devices stmt).
- 11 new character sets added (see NASC stmt).
- Previous optional bar codes now standard.
- Printer setup via bar code wand introduced.

Compatibility:
- Font names for bitmap fonts translate to corresponding scalable font.
- Device "ram:" translates to "c:".
- Deleted commands will be ignored – no error conditions occur.

Deleted Instructions:

PRINTFEED NOT	Has no meaning in UBI Fingerprint 7.11.
REMOVE FONT	Has no meaning in UBI Fingerprint 7.11 (bitmap fonts no longer used).
RIBBON SAVE ON/OFF	No ribbon save device exists for UBI Fingerprint 7.11 compatible printers.
STORE	Obsolete. Replaced by STORE INPUT.

Modified instructions:

BARFONT	Supports Unicoded TrueDoc and TrueType fonts with scaling and slanting.
CHDIR	Supports new memory devices
DEVICES	Removed: "uart3:", "cutter:", "ram:", "prel:", "rs485:", "msg:", "par:", "bscrypt:", "null:" and "ind:". New: "c:", "lock:", "storage:", "tmp:", "wand:".
FILES	Possible to include/exclude system files.
FONT	Supports Unicoded TrueDoc and TrueType fonts with scaling and slanting.
FORMAT	Possible to include/exclude system files.
FRE	Now returns the number of free bytes in the temporary memory.
FUNCTEST	Parameter RAM deleted, parameter KERNEL added.
FUNCTEST$	Parameter RAM deleted, parameter KERNEL added.
IMAGELOAD	Supports downloading of both images and fonts.
NASC	11 new single-byte character sets can be selected.
OPTIMIZE ON/OFF	Optimizing strategies “PRINT” and “STRING” no longer supported.
PORTIN	Now supports 8 in ports and 12 out ports.
PORTOUT ON/OFF	Now supports 12 out ports.
SETUP	Some setup parameters changed, deleted or added.
SYSVAR	Some system variables deleted or added.
TESTFEED	TESTFEED is now the only method for adjusting the label stop sensor.
VERSION$	Now returns somewhat different information.

New Instructions:

FONTD	Selects double-byte fonts.
NASCD	Selects double-byte character sets.

2. Getting Started

2.1 Computer Connection

The UBI Fingerprint firmware is stored in a Flash SIMM on the printer's CPU board. No floppy disks or operative system, like e.g. MS-DOS, is required. The printer only needs to be connected to a mains supply.

Unless the printer is fitted with a program that allows it to be used independently ("stand-alone"), you must also connect it to some kind of device, which can transmit characters in ASCII format. It can be anything from a non-intelligent terminal to a mainframe computer system.

For programming the printer, you need a computer with a screen and an alphanumeric keyboard, that provides two-way serial communication, preferably using RS 232C, (e.g. an IBM PC or similar computer with Microsoft Windows 3.11[1]). Use e.g. Windows Notepad or Write for writing programs and Windows Terminal for communication with the printer.

Connect the printer and host as described in the Installation & Operation manual for the printer model in question. If the printer has several communication ports, it is recommended to use the serial port "uart1:" for programming, which by default is set up for RS 232C. Other optional serial communication ports could also be used.

[1]/. Although most examples in this manual assumes a host running MS Windows 3.11, other operative systems can also be used, e.g. Windows 95, Windows NT, DOS, Macintosh OS, OS-2 etc, as long you have a terminal program that can communicate with the printer and some kind of word processing program.

It is possible to set up the printer's communication protocol to fit the host computer. However, until you have become familiar with the UBI Fingerprint concept, it may be easier to adapt the host to the printer's default setup parameters:

Default communication setup on "uart1:"

• Baud rate:	9600
• Character length:	8
• Parity:	None
• No. of stop bits:	1
• Flow control:	XON/XOFF to and from host
• New line:	CR/LF (Carriage Return + Line Feed)

☞ ***Communication Setup***
Also see:
• Chapter 15.6
• Installation & Operation manual

2.2 Check Paper Supply

Check that the printer has an ample supply of paper or other receiving material and, when applicable, of thermal transfer ribbon. Refer to the Operator's Guide or the User's Manual for loading instructions.

☞ ***Paper and Ribbon Load***
Also see:
• User's Guide
• Installation & Operation manual

2.3 Turn On the Printer

Check that the printhead is lowered. Turn on the main switch, which is fitted on the printer's rear plate and check that the "Power" control lamp comes on. Then watch the display window. What happens next depends on what kind of startup file there is in the printer.

> **WARNING!**
> **Make sure that any paper cutter is locked in closed position.**
> The cutter may be activated when the power is turned on!

After a short while, when the printer has performed certain self-diagnostic tests and loaded the startup program, a countdown menu will usually be displayed:

```
ENTER=UBI SHELL
5 sec.    v.4.x
```

2.4 UBI Shell Startup Program

☞ ***UBI Shell Startup Program***
Also see:
• Installation & Operation manual

The countdown menus indicate that the printer is fitted with one of the UBI Shell startup programs. Wait until the 5 seconds countdown is completed. Then, by default, this menu will be displayed:

```
UBI Fingerprint
7.xx
```

This or similar messages indicates that the printer has entered the immediate mode of UBI Fingerprint, where you can start your programming. Please proceed at chapter 2.8.

If the UBI Shell countdown menus are shown, but are followed by any other message than "UBI Fingerprint 7.xx", some other application has already been selected in UBI Shell. Refer to the Installation & Operation manual for information on how to select the UBI Fingerprint option.

2.5 No Startup Program

If the printer is not fitted with any startup program at all, the display window should show the following message directly after power-up:

```
UBI Fingerprint
7.xx
```

This means that the printer has entered the immediate mode of UBI Fingerprint. Proceed at chapter 2.8.

2.6 Custom-Made Startup Program

If any other kind of message is displayed than those illustrated above, the printer is provided with some kind of custom-made startup program, which you must break before you can start programming.

- Go on to chapter 2.7.

2.7 Breaking a Startup Program

☞ ***Breaking a Program***
Also see:
• Chapter 5.12

Default Method (break from keyboard)

- Press the <**C**> key and keep it pressed down while also pressing the <**Pause**> key.

Other Methods

- The program may be provided with other means for breaking the program, e.g. by sending a certain character from the host or by pressing another key or combination of keys. Break from keyboard may also be disabled completely.

When a break interrupt has been executed and you have entered the immediate mode, there will be no change in the printer's display, but a message should appear on the screen of the host, provided you have a working two-way communication:

```
User break in line XXXX
```

How to go on

- If you have succeeded in breaking the program, proceed at chapter 2.8.

2.8 Communications Test

Check that you have entered the immediate mode and have a working two-way serial communication by sending a simple instruction from the host to the printer. On the keyboard of the host, type:

? VERSION$ ↵ *(↵ = Carriage Return key)*

> ☞ ***Version Check***
> Also see:
> • Chapter 15.11

The printer should respond immediately by returning the version of the installed UBI Fingerprint software to the screen of the host, e.g.:

```
UBI Fingerprint 6.11
Ok
```

> ☞ ***Communication Setup***
> Also see:
> • Chapter 15.6
> • Installation & Operation manual

This indicates that the communication is working both ways.

If the communication does not work, turn off the printer and check the connection cable. Also check if the communication setup in the host corresponds to the printer's setup and if the connection is made between the correct ports. Check the verbosity level. Then try the communication test again.

> ☞ ***Verbosity***
> Also see:
> • Chapter 7.7
> • Chapter 15.7

Another possible cause of error may be that another communication channel than "uart1:" has been selected for UBI Fingerprint in UBI Shell. Reselect the UBI Fingerprint application for "uart1:" as described in the Installation & Operation manual.

> ☞ ***UBI Shell***
> Also see:
> • Installation & Operation manual

Once you know that the communication is working, you may go on and make the printer auto-adjust its paper feed according to the type of labels loaded. Simultaneously press the <**Shift**> and <**Feed**> keys on the printer's built-in keyboard. The printer will feed out at least two blank labels (or corresponding).

> ☞ ***Text Field Printing***
> Also see:
> • Chapter 10.2

Finally send a line of text to make sure that characters transmitted from the terminal are interpreted as expected by the printer's software:

```
FONT "Swiss 721 BT" ↵
PRTXT "ABCDEFGHIJKLM" ↵
PRINTFEED ↵
```

Each line will be acknowledged by an “Ok” on the screen, provided that it has been entered correctly, that there is a working two-way serial communication, and that the verbosity is on. When you press the “Carriage Return” key the third time, the printer will feed out a label, ticket, tag or piece of strip with the text printed near the lower left corner of the printable area.

> ☞ ***Character Sets***
> Also see:
> • Chapter 9.1
> • UBI Fingerprint Reference Manual

Try using other characters between the quotation marks in the third line, especially typical national characters like ÅÄÖÜ¿¢¥ç etc. Should any unexpected characters be printed, you may need to select another character set, see **NASC** statement in chapter 9.1, or switch from 7-bit to 8-bit communication.

3. Creating a Simple Label

3.1 Introduction

To get a quick impression of how UBI Fingerprint works, start by creating a simple label following the step-by-step instructions below. Later in this manual, the various functions will be explained in greater detail. You can also look up the instructions in the UBI Fingerprint Reference Manual.

☞ *Carriage Return Character*
Also see:
• Chapter 4.1

Use a word processing program, e.g. Windows Notepad, to enter the program lines. Use a space character to separate the line number from the instruction that follows. Finish each line with a carriage return character, indicated by ø below.

When you have entered a batch of program lines, copy the lines and paste them into a communication program, e.g. Windows Terminal, which is connected to the printer (see chapter 2.11).

The printer will not execute the program until you have entered **RUN** + Carriage Return.

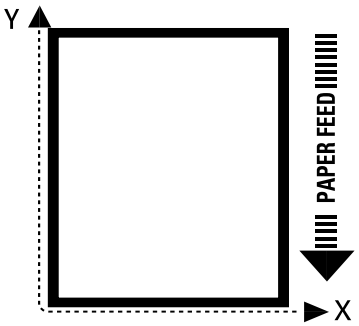
3.2 Printing a Box

Let us start by printing a box 430 dots high and 340 dots wide with a line thickness of 15 dots. The box is inserted at position X=10, Y=10:

```
NEW
10    PRPOS 10, 10 ↵
20    PRBOX 430,340,15 ↵
200   PRINTFEED ↵
300   END ↵
RUN ↵
```

Note: The printer will not execute the program until you have typed RUN ↵.

☞ *Box Field Printing*
Also see:
• Chapter 10.5



Note:
This example is designed to be run on any present UBI Fingerprint 7.xx-compatible EasyCoder printer connected to a terminal or computer and loaded with a paper web (preferably labels) according to the following specifications.

Label size:

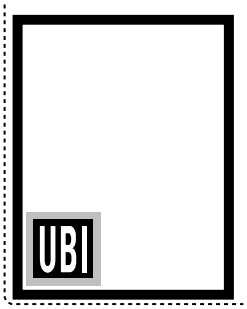
Width:	*≥ 52.8 mm*	*(2.08")*
Length:	*≥ 70 mm*	*(2.75")*

3.3 Printing an Image

Now we add the image "UBI.1" after changing the position coordinates to X=30,Y=30.

```
30    PRPOS 30,30 ↵
40    PRIMAGE "UBI.1" ↵
RUN ↵
```

☞ *Image Field Printing*
Also see:
• Chapter 10.4

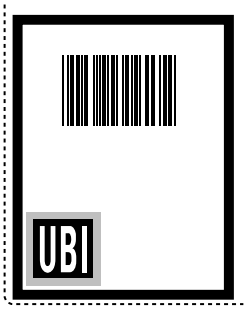


3.4 Printing a Bar Code

Before you print a bar code, you need to choose a bar code type. Note there is no blank space in the bartype name.

```
50    PRPOS 75,270 ↵
60    BARTYPE "CODE39" ↵
70    PRBAR "UBI" ↵
RUN ↵
```

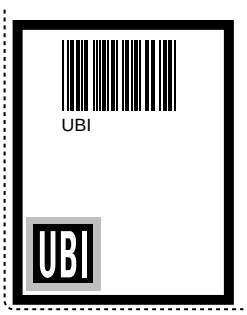
☞ *Bar Code Field Printing*
Also see:
• Chapter 10.3



3.5 Printing Human Readables

To get human readable text printed under the bar code, add these lines:

```
1     BARFONT ON ↵
2     BARFONT "Swiss 721 BT", 6 ↵
RUN ↵
```



3.6 Printing Text

Add a line of text at position X=25,Y=220:

```
80    PRPOS 25,220 ↵
90    FONT "Swiss 721 BT", 6 ↵
100   PRTXT "My FIRST Label!" ↵
RUN ↵
```

☞ *Text Field Printing*
Also see:
• Chapter 10.2

3.7 Listing the Program

To view the whole program, type:

```
LIST ↵
```

The lines will be listed in ascending order on your terminal's screen:

```
1     BARFONT ON
2     BARFONT "Swiss 721 BT", 6
10    PRPOS 10,10
20    PRBOX 430,340,15
30    PRPOS 30,30
40    PRIMAGE "UBI.1"
50    PRPOS 75,270
60    BARTYPE "CODE39"
70    PRBAR "UBI"
80    PRPOS 25,220
90    FONT "Swiss 721 BT", 6
100   PRTXT "My FIRST label!"
200   PRINTFEED
300   END
ok
```

☞ *Program Editing and Listing*
Also see:
• Chapter 5.4

3.8 Changing a Program Line

If you want to change a program line, simply rewrite the line using the same line number. For example, move the text to the right by rewriting line number 80 with new coordinates:

```
80    PRPOS 75,220 ↵
RUN ↵
```

3.9 Saving the Program

☞ ***Saving***
Also see:
• Chapter 5.13

If you want to save your first attempt, issue the following instruction:

```
SAVE "LABEL1" ↵
```

Your program will be saved in the printer's memory under the name:

```
LABEL1.PRG
```

3.10 Error Handling

☞ ***ERRHAND.PRG***
Also see:
• Chapter 16.4

The program above is very simple and there is a very small risk of encountering any errors. When writing more complex programs, you might find use for an errorhandler. For that purpose we have included a program called ERRHAND.PRG in the firmware. Should your printer not contain any errorhandling program, you will find ERRHAND.PRG listed in chapter 16.4.

ERRHAND.PRG contains subroutines that e.g. displays the type of error on the printer's LCD display (e.g. "OUT OF PAPER" or "HEAD LIFTED"), prints the error number on your screen, and assigns subroutines to some of the keys on the keyboard (if any). There is also a subroutine that performs a **PRINTFEED** with error-checking. The ERRHAND.PRG occupies lines 10, 20 and 100000–1900000.

3.11 Renumbering Lines

☞ ***Renumbering Program Lines***
Also see:
• Chapter 5.4

If ERRHAND.PRG is merged with the program you just wrote, lines 10 and 20 in your program will be replaced with lines 10 and 20 from ERRHAND.PRG. Therefore you have to renumber your program, so that your program begins with an unoccupied number, e.g. 50, before ERRHAND.PRG is merged:

```
RENUM 50,1,10 ↵
Ok

LIST ↵
50    BARFONT ON
60    BARFONT "SW030RSN"
70    PRPOS 10,10
80    PRBOX 400,300,10
90    PRPOS 25,25
100   PRIMAGE "UBI.1"
110   PRPOS 75,250
120   BARTYPE "CODE39"
130   PRBAR "UBI"
140   PRPOS 25,200
150   FONT "SW030RSN"
160   PRTXT "My FIRST label!"
170   PRINTFEED
180   END
ok
```

3.12 Merging Programs

☞ *Merging programs*
Also see:
• Chapter 6.3

Now your label-printing program LABEL1.PRG will not interfere with ERRHAND.PRG and you can merge the two programs into a single program. In fact, you will create a copy of ERRHAND.PRG which is merged into LABEL1.PRG. Thus the original ERRHAND.PRG can be merged into more programs later:

```
MERGE "rom:ERRHAND.PRG" ↵
```

3.13 Using the Print Key

☞ *Branching and Loops*
Also see:
• Chapter 5.6 (GOTO)
• Chapter 5.7 (GOSUB)

Instead of using a **PRINTFEED**statement, we will use a subroutine in ERRHAND.PRG. Because ERRHAND.PRG assigns functions to e.g. the <**Print**> key, you can create a loop in the program so you will get a label every time you press the <**Print**> key.

```
160   GOSUB 500000 ↵
170   GOTO 170 ↵
RUN ↵
```

Try pressing different buttons on the printer's keyboard. Only those, to which functions been assigned in ERRHAND.PRG (i.e. the <**Pause**>, <**Print**>, <**Setup**> and <**Feed**> keys) will work.

You can break the program by simultaneously pressing the <**Shift**> and <**Pause**> keys.

Save the program again using the same name as before:

```
SAVE "LABEL1" ↵
```

The previously saved program "LABEL1.PRG" will be replaced by the new version.

With this example, we hope you have got a general impression of the basic methods for UBI Fingerprint programming and that you also see the advantages of using ERRHAND.PRG or a similar program for errorhandling and initiation.

ERRHAND.PRG can easily be modified to fit into more complex programs and we recommend that you use it when writing your programs until you feel ready to create errorhandling programs yourself (see chapter 16 “Error-Handling”).

4. Terminology and Syntax

4.1 Lines

You will always use one or several lines to give the instructions to the printer, regardless whether you work in the immediate mode, in the programming mode, or in the UBI Direct Protocol. The difference is that in the programming mode, the lines are always numbered (visibly or invisibly), whereas in the immediate mode and the UBI Direct Protocol, they must not be numbered.

Note:
If you enter a carriage return on your terminal, the printer will, by default, echo back a Carriage Return + a Line Feed (ASCII 13 + 10 decimal). Using the setup option "New Line", you may restrict the printer only to echo back either a Carriage Return (ASCII 13 dec.) or a Line Feed (ASCII 10 dec.).

A line may contain up to 300 characters. A line must always be terminated by a Carriage Return character (ASCII 13 decimal), see note. When the line reaches the right edge of the screen of the host, it will usually wrap to the next screen line.

Theoretically, line numbers up to > 2 billion can be used. If you choose to enter the line numbers manually, start by numbering the lines from 10 and upwards with an increment of 10, i.e. 10, 20, 30, 40 etc. That makes it possible to insert additional lines (e.g. 11,12,13...etc.), when the need arises. However, the line numbers are your own decision, since you must type them yourself.

☞ ***Programming Mode***
Also see:
• Chapter 5.4

You can also omit line numbers at edition and let the software number the lines automatically. Such line numbers will not be visible before the program is listed.

After having typed the line number, use a blank space to separate it from the statement or function that follows. That makes it easier to read the program without having to list it.

Several instruction may be issued on the same line, provided they are separated by colons (:), e.g.:

☞ ***Immediate Mode***
Also see:
• Chapter 5.3

```
100  FONT "Swiss 721 BT":PRTXT "HELLO"
```

This is especially useful in the immediate mode (see chapter 5.3) and in the UBI Direct Protocol, where you can send a complete set of instructions as a single line, e.g.:

☞ ***UBI Direct Protocol***
Also see:
• UBI Direct Protocol 7.xx, Programmer's Guide

```
PP100,250:FT"Swiss 721 BT":PT"Text 1":PF ↵
```

It is not possible to alter a line after it has been transmitted to the printer. If you want to change such a line, you must send the whole line again using the same line number, or delete it using a **DELETE** statement (see chapter 5.4).

A statement is an instruction, which specifies an operation. It consists of a keyword (e.g. **PRTXT**), usually followed by one or several parameters, flags, or input data, which further define the statement.

The keyword can be entered as uppercase or lowercase letters but will always appear as uppercase letters, when the program is listed on the screen of the host. Some keywords can be used in an abbreviated form, e.g. **PRTXT** may also be entered as **PT**.

4.2 Statements

☞ ***Keywords***
Also see:
• Chapter 4.8

You may use a blank space to separate the keyword from the rest of the statement, which must be entered exactly according to the specified syntax. Note that in some cases, a space character is a compulsory part of the keyword, e.g. **`LINE_INPUT`**. When such is the case, it is indicated by the syntax description in the UBI Fingerprint Reference Manual.

4.3 Functions

☞ ***Operators***
Also see:
• Chapter 4.9

A function is a procedure, which returns a value. A function consists of a keyword combined with values, flags, and/or operators. The keyword can be entered as uppercase or lowercase letters, but it will always appear as uppercase letters, when the program is listed on the screen. Values, flags, and operators must be enclosed by parentheses ( ). The operators will be explained later on.

Examples:

`CHR$(65)`	*Keyword with parameter.*
`TIME$("F")`	*Keyword with flag.*
`ABS (20*5)`	*Keyword with arithmetic operator (*) and values.*
`IF(PRSTAT AND 1)...`	*Keywords, logical operator (AND) and value.*

☞ ***Conditional Instructions***
Also see:
• Chapter 5.5

A function can be entered inside a statement or on a line containing other instructions. They are often used in connection with conditional statements, e.g.:

```
320  IF (PRSTAT AND 1) THEN GOTO 1000
```

Blank spaces may be inserted to separate the function from other instructions and also to separate the keyword from the rest of the statement.

4.4 Other Instructions

In addition to statements and functions, there are a few other types of specialized instructions such as the **`DATE$`** and **`TIME$`** variables, the **`SYSVAR`** system array and the **`RUN pcx2bmp`** command, which do not fit into the above-mentioned categories.

4.5 Expressions

In the descriptions of the syntax for the various instructions, the word “Expression” is used to cover both constants and variables.

Expressions are of two kinds:

- **String expressions** are carriers of alphanumeric text, i.e. string constants and string variables.
- **Numeric expression**s contain numeric values, numeric variables and operators only, i.e. numeric constants and numeric variables.

4.6 Constants

Constants are fixed text or values. There are two kinds:

- **String constants** are sequences of characters, i.e. text. If digits or operators are included, they will be considered as text and will not be processed. String constants must always be started and terminated by double quotation marks ("..."), e.g. "TEST.PRG".
- **Numeric constants** are fixed numeric values. Only decimal integers are allowed, i.e. 1, 2, 3, 4, 5 etc. Decimal points (e.g. 1.56890765) are not supported. Values may be positive or negative. Positive number may optionally be indicated by a leading plus sign (+), whereas negative numbers always must be indicated by a leading minus sign (-).

Note that certain characters, e.g. digits, can be either string constants (text) or numeric constants (numbers). To allow the software to detect that difference, string constants must always be enclosed by double quotation marks (" ....."), as opposed to numeric constants.

4.7 Variables

Variables are value holders. There are two main types:

- **String variables** are used to store strings entered as string constants or produced by UBI Fingerprint instructions. Max. size is 64 kbytes. String variables are indicated by a trailing $ sign.

  Examples:

```
A$ = "UBI PRINTER"
B$ = TIME$
LET C$ = DATE$
```

- **Numeric variables** are used to store numbers, entered as numeric constants, or produced by UBI Fingerprint instructions or operations. Numeric variables are indicated by a trailing % sign.

  Examples:

```
A% = 150
B% = DATEDIFF ("981001","981130")
LET C% = 2^2
```

The name of a variable may consist of letters, numbers and decimal points. The first character must always be a letter. No keywords or keyword abbreviations must be used. However, completely embedded keywords are allowed.

Examples:

LOC	*is a keyword*
CLOCK$ = "ABC"	*is OK*
LOC$ = "ABC"	*causes an error*
LOCK$ = "ABC"	*causes an error*

The presently used keywords and keywords reserved for future program enhancement are listed on next page.

4.8 Keyword List

#	BT	FONTNAME$	LIST	PORTOUT	SOUND
'	BUSY	FONTS	LOAD	PP	SPACE$
(	CHDIR	FOR	LOC	PRBAR	SPC
)	CHECKSUM	FOR APPEND AS	LOCATE	PRBOX	SPLIT
*	CHR$	FOR INPUT AS	LOF	PRIMAGE	STEP
+	CLEANFEED	FOR OUTPUT AS	LSET	PRINT	STOP
,	CLEAR	FORMAT	LTS&	PRINT USING	STORE
-	CLL	FORMFEED	MAG	PRINTFEED	STR$
/	CLOSE	FRE	MAP	PRINTONE	STRING$
:	COM ERROR	FT	MERGE	PRLINE	SWAP
;	COMBUF$	FUNCTEST	MID$	PRPOS	SYSTEM
<	COMSET	GET	MOD	PRSTAT	SYSVAR
<=	COMSTAT	GOSUB	NAME	PRTXT	TAB
<>	CONT	GOTO	NASC	PT	TESTFEED
=	COPY	HEAD	NASCD	PUT	THEN
=<	COUNT&	HEX$	NEW	PX	TICKS
=>	CSRLIN	HOLIDAY$	NEXT	RANDOM	TIME$
>	CSUM	IF	NI	RANDOMIZE	TIMEADD$
><	CUT	II	NORIMAGE	READ	TIMEDIFF
>=	DATA	IMAGE	NOT	READY	TO
?	DATE$	IMAGENAME$	OFF	REBOOT	TRANSFER
ABS	DATEADD$	IMAGES	OFF LINE	REDIRECT OUT	TRANSFER$
ACTLEN	DATEDIFF	IMMEDIATE	ON	REM	TRANSFERSET
ALIGN	DELETE	IMP	ON BREAK	REMOVE	TROFF
AN	DEVICES	INKEY$	ON COMSET	RENUM	TRON
AND	DIM	INPUT	ON ERROR GOTO	RESET	VAL
AS	DIR	INPUT$	ON KEY	RESTORE	VERBOFF
ASC	ELSE	INSTR	ON LINE	RESUME	VERBON
BARADJUST	END	INT	OPEN	RESUME NEXT	VERSION$
BARFONT	EOF	INVIMAGE	OPT	RETURN	WEEKDAY
BARHEIGHT	EQV	IP	OPTIMIZE	RIBBON	WEEKNUMBER
BARMAG	ERL	KEY	OR	RIGHT$	WEND
BARRATIO	ERR	KEYBMAP$	PB	RND	WHILE
BARSET	FF	KILL	PEC2DATA	RSET	WRITE
BARTYPE	FIELD	LAYOUT	PEC2LAY	RUN	XOR
BEEP	FIELDNO	LBLCOND	PECTAB	SAVE	XYZZY
BF	FILE&	LED	PF	SET FAULTY DOT	\
BH	FILES	LEFT$	PL	SETSTDIO	^
BM	FIX	LEN	PLAY	SETUP	
BR	FONT	LET	PM	SGN	
BREAK	FONTD	LINE INPUT	PORTIN	SORT	

4.9 Operators

There are three main types of operators – arithmetic, relational, and logical:

Arithmetic Operators (integers only)

+	Addition	(e.g. **2+2=4**)
-	Subtraction	(e.g. **4-1=3**)
*****	Multiplication	(e.g. **2*3=6**)
****	Integer division	(e.g. **6\2=3**)
MOD	Modulo arithmetic (results in an integer value which is the remainder of an integer division, e.g. **5MOD2=1**)	
^	Exponent	(e.g. **5^2=25**)

Parentheses can be used to specify the order of calculation, e.g.:

```
7+5^2\8 = 10
(7+5^2)\8 = 4
```

Relational Operators

<	less than
<=	less than or equal to
<>	not equal to
=	equal to (also used as an assignment operator)
>	greater than
>=	greater than or equal to

Relational operators return:

-1	if relation is TRUE
0	if relation is FALSE

The following rules apply:

- Arithmetic operations are evaluated before relational operations.
- Letters are greater than digits.
- Lowercase letter are greater than their uppercase counterparts.
- The ASCII code "values" of letters increase alphabetically and the leading and trailing blanks are significant.
- Strings are compared by their corresponding ASCII code value.

Logical Operators

AND	conjunction
OR	disjunction
XOR	exclusive or
EQV	equivalent

Logical operators combine simple logical expressions to form more complicated logical expressions. The logical operators operate bitwise on the arguments, e.g.:

```
1 AND 2 = 0
```

Logical operators can be used to connect relational operators, e.g.:

```
A%10 AND A%<100
```

Logical operators can also be used to mask bits, e.g.:

```
A%=A% AND 128
```

4.9 Operators, cont'd.

The principles are illustrated by the following tables, where A and B are simple logical expressions.

Logical operator: AND

A	B	A **AND** B
1	1	1
1	0	0
0	1	0
0	0	0

Logical operator: XOR

A	B	A **XOR** B
1	1	0
1	0	1
0	1	1
0	0	0

Logical operator: OR

A	B	A **OR** B
1	1	1
1	0	1
0	1	1
0	0	0

Logical operator: EQV

A	B	A **EQV** B
1	1	1
1	0	0
0	1	0
0	0	1

4.10 Devices

"Device" is a generic term for communication channels, various parts of the printer's memory, and operator interfaces such as the printer's display and keyboard.

Name	No.	Can be OPENed for...	Remarks
Communication:			
console:	0	Input/Output	Printer's display and/or keyboard
uart1:	1	Input/Output	Serial communication port
uart2:	2	Input/Output	Serial communication port (option)
centronics:	4	Input	Parallel communication
Memory:			
rom:	N/A	Input (files only)	Printer's firmware (Kernel) plus read-only memory card
c:	N/A	Input/Output/Random	(alternative name "ram:")
temp:	N/A	Input/Output/Append/ Random (files only)	Printer's temporary memory
card1:	N/A	Input/Output/Append/ Random (files only)	SRAM memory card
Special:			
lock:	N/A	Input	Electronic keys
storage:	N/A	Input/Output/Random	Electronic keys
wand:	N/A	Input	Data from Code 128 bar code via printer's bar code wand interface

☞ ***Files***
Also see:
• Chapter 6 (File system)
• Chapter 7 (Input, Append, Random)
• Chapter 8 (Output, Random)

The devices can be listed by means of a **DEVICES** statement. All devices will be listed regardless if they are installed or not.

Devices are referred to by name in connection with instructions concerning directories (e.g. **SAVE**, **KILL**, **FORMAT**) and with **OPEN** statements. Note that the names of all devices should end with a colon (:) and the name should be enclosed by double quotation marks, e.g. "tmp:". Use lowercase characters only in device names.

In instructions used in connection with communication (e.g. **BREAK**, **BUSY/READY**, **COMSET**), the keyboard/display unit and the communication channels are specified by numbers instead of names:
0 = "console:"
1 = "uart1:"
2 = "uart2:"
4 = "centronics:"

5. UBI Fingerprint Programming

5.1 Introduction

The UBI Fingerprint 7.xx firmware works in two main modes, the "Immediate Mode" and the "Programming Mode". A special case is the UBI Direct Protocol 7.xx, which is described in a separate Programmer's Guide and will not be explained any further in this manual.

Immediate Mode implies that the instructions are executed at once as soon as a carriage return is received. Most instructions can be used, but the instructions cannot be saved after execution.

Programming Mode is used to enter instructions in the form of program lines. The lines can be manually provided with visible line numbers at editing, or be automatically provided with invisible line numbers by the printer's software. No execution is performed until a **RUN** statement is issued in the Immediate Mode, i.e. on a line without number. The program can be saved in the printer's memory and used again.

5.2 Editing Methods

☞ ***Computer Connection***
Also see:
• Chapter 2.1

To be able to program a printer, you need a terminal or host computer with a screen and a keyboard and a working two-way serial communication between printer and host, preferably RS 232C on communication channel "uart1:". The host must be able to transmit and receive ASCII characters, e.g. by means of a communication program like Windows Terminal.

There are three main methods of writing and transmitting a program to the printer:

- **Line-by-Line Method**
  If you have an "non-intelligent" terminal that just can transmit and receive ASCII characters, you must write and send each line separately.

  ☞ ***Verbosity***
  Also see:
  • Chapter 7.7
  • Chapter 15.7

  Each line will be checked for possible syntax errors as soon as the printer receives it and the printer will return either "Ok" or an error message to the screen of the host, provided verbosity is on.

  ☞ ***Error Messages***
  Also see:
  • Chapter 16.1

  If you need to correct a mistake, you must rewrite the complete line using the same line number. Thus, this method is not suited for the programming without line numbers.

  Note that even if most examples of computer connection in this manual assumes a PC running under MS Windows (3.11, Win 95 or NT4), UBI Fingerprint is by no means restricted to such computers. Other personal computers and operating systems, such as DOS, Apple Macintosh OS, OS-2, Unix etc., as well as larger computer systems, can be used following the same principles.

5.2 Editing Methods, cont'd.

☞ ***Verbosity***
Also see:
• Chapter 7.7
• Chapter 15.7

☞ ***Error Messages***
Also see:
• Chapter 16.1

- **Copy-and-Paste Method**
  If the host computer is fitted with both a communication program (e.g. Windows Terminal) **and** a word-processing program (e.g. Windows Write or Windows Notepad), you can write the program, partly or completely, in the word processor and then Copy and Paste it into the communication program.

  Each line will be checked for possible syntax errors as soon as the printer receives it and the printer will return an error message after each line where an error has been detected, provided verbosity is on.

  If you need to correct a mistake, you can make the correction in the word processor and then copy and paste the line into the communication program. If you do not use line numbers, you must Copy and Paste the complete corrected program back to the communication program.

- **Send Text Method**
  If the host computer is fitted with both a communication program (e.g. Windows Terminal) and a word-processing program (e.g. Windows Write or Windows Notepad), you can write the program, partly or completely, in the word processor and send the whole text file to the printer by means of the communication program (e.g. "Transfers; Send Text File" in Windows Terminal).

  Each line will be checked for possible syntax errors as soon as the printer receives it and the printer will return an error message after each line where an error has been detected, provided verbosity is on.

  If you need to correct a mistake, you can make the correction in the word processing program and then send the complete program again via the communication program.

5.3 Immediate Mode

The Immediate Mode can be used for four main purposes:
- Printing of labels that you will never need to print again.
- Printing of labels, which have been edited and saved in the host computer and are downloaded as text strings to the printer.
- Editing of programs to be executed in the programming mode.
- Issuing of instructions outside the execution of programs in the programming mode, e.g. **DELETE**, **LOAD**, **MERGE**, **NEW**, **REBOOT** or **RUN**.

Rather than creating programs in the Programming Mode, in some cases you may want to edit the label in your host computer and transmit the printing instructions and data to the printer in the form of text strings. This method resembles the so called "Escape sequences" used in earlier generations of label printers.

5.3 Immediate Mode, cont'd.

To make the strings shorter, use the UBI Fingerprint abbreviations. Several statements can be issued on the same line separated by colons (:), on separate lines, or using a mix of both methods.

Examples:
All intructions can be issued in a single line....

```
PP160,250:DIR3:AN4:FT"Swiss 721 BT":PT"Hello":PF ↵
```

or with each instruction as a separate line...

```
PP160,250 ↵          (print start position)
DIR3 ↵               (print direction)
AN4 ↵                (alignment)
FT"Swiss 721 BT" ↵   (font select)
PT"Hello" ↵          (text input data)
PF ↵                 (print one copy)
```

☞ ***Standard Error-Handling***
Also see:
• Chapter 16.1

As soon as a carriage return is received, the software checks the instructions for syntax errors. Provided there is a working two-way communication and the verbosity is on, the printer will either return an error message or "Ok" to the host.

☞ ***UBI Direct Protocol***
Also see:
• UBI Direct Protocol 7.xx, Programmer's Guide

This type of communication works well and is easy to learn, but it does not take full advantage of the flexibility and computing capacity offered by the UBI Fingerprint printers. For example, you cannot save the labels in the printer but must download each new label, and all error-handling must be taken care of by the host.

Rather than using the Immediate Mode, the UBI Direct Protocol is usually to prefer, since it allows variable input data to be combined with predefined layouts, handles counters and contains a flexible error-handler.

Beside printing text, bar codes and graphics, you can perform other tasks in the Immediate Mode as well, e.g. calculation. Try typing this instruction on the keyboard of the host:

```
? ((5^2+5)\3)*5 ↵        (↵ = Carriage Return key)
```

The calculation will be performed immediately and the result will be returned to the screen of the host:

```
50
Ok
```

Important:
To send an instruction from the terminal to the printer, press the Carriage Return key. In the programming examples later on in this manual, this character will be omitted, but you must not forget to enter it via the keyboard of the host.

Four keys or key combinations are enabled in the Immediate Mode, obviously provided that the printer is fitted with the key(s) in question:

- The <**Print**> key or button produces a **FORMFEED** operation.
- The <**Feed**> key produces a **FORMFEED** operation.
- The <**Shift**> + <**Feed**> keys produce a **TESTFEED** operation.
- The <**Setup**> key gives access to the Setup Mode.

5.3 Immediate Mode, cont'd.

When the printhead is lowered and the <**Print**> or <**Feed**> keys are pressed, three possible error conditions can cause an error message in English to be displayed:

- "Error 1005 -Press any key!-" (Out of paper)
- "Error 1031 -Press any key!-" (Next label not found)
- "Error 1027 -Press any key!-" (Out of ribbon)

After the error has been attended to, the error message can be cleared by pressing any of the above-mentioned keys.

When the printhead is lifted, the <**Print**> and <**Feed**> keys will run the printers mechanism in order to facilitate cleaning of the print roller, i.e. the rubber-coated roller that drives the paper forward under the printhead. The motor will stop automatically when the print roller has completed a few rotations.

5.4 Programming Mode

The Programming Mode is used to execute instructions entered in the form of program lines. The firmware assumes input to the Programming Mode in two cases:

- When a line starts with a number.
- After an **IMMEDIATEOFF** statement has been executed. (See *"Programming without Line Numbers"* later in this chapter).

One or several lines make up a program, which can be executed as many times as you wish. A program can also be saved, closed, copied, loaded, listed, merged, and killed, see chapter 6.3. All lines have line numbers, that are either manually entered when the program is edited, or provided automatically and invisibly by the firmware when an **IMMEDIATEON** statement has been executed.

Each time the printer receives a program line followed by a Carriage Return character, the firmware checks the line for possible syntax errors. If an error is encountered, an error message will be returned to the host, provided there is a working two-way communication and the verbosity is on.

The program is executed in ascending line number order when a **RUN** statement is issued in the Immediate Mode, i.e. on a line without any line number. However, various types of branching and loops can be created in the program that makes the execution deviate from a strict ascending order.

☞ ***Autoexec-files (startup files)***
Also see:
• Chapter 5.13

Note that the editing of the program takes place in the Immediate Mode, while the execution is performed in the Programming Mode. Often, programs are made as autoexec (startup) files that start up automatically when the printer is turned on, and keeps on running infinitely.

5.4 Programming Mode, cont'd.

Programming with Line Numbers

In this case you will start each line by manually entering a line number. We recommend that you start with line number 10 and use an increment of 10 between lines to allow additional lines to be inserted later. To make the program easier to read, you can use a space character between the line number and the instruction. If not, the software will insert a space character automatically when the program is listed. Let us use the calculation example from the Immediate Mode. It would look like this in the Programming Mode:

Important:
To send an instruction from the terminal to the printer, press the Carriage Return key. In the programming examples later on in this manual, this character will be omitted, but you must not forget to enter it via the keyboard of the host.

```
10 ? ((5^2+5)\3)*5 ↵
RUN ↵
```

yields:

```
15
Ok
```

Let us have a look at the lines:

- The first line consists of a line number (10) followed by an optional space character and the instruction **`? ((5^2+5)\3)*5`** (**`?`** is a shorthand form for the statement **`PRINT`**, which returns the result of the calculation to the screen of the host). The line is terminated by a Carriage Return character.
- Next line has no line number, and contains the statement **`RUN`**, which orders the printer to execute all preceding numbered lines in consecutive ascending order according to their line numbers.
- The result (15) will be displayed on the terminal's screen followed by "Ok" to indicate that execution was successful.

In this manual, the programming examples will generally have line numbers in order to make them easier to understand. For more complex programs, programming without line numbers, as explained on next page, may be both easier and quicker.

5.4 Programming Mode, cont'd.

Programming without Line Numbers

You can choose to omit entering line numbers manually when writing a program. This is a special case of the Programming Mode, but in order to make the printer understand what you want to do, you must turn off the Immediate Mode by means of an **IMMEDIATE OFF** statement. (Normally, the software interprets the lack of line numbers as Immediate Mode).

Then you can write the program line by line without having to type a line number at the start of each line. In other respects, you can generally work just as in the normal programming mode.

☞ ***Branching the Program Execution***
Also see:
• Chapter 5.6 – 5.8

However, a major difference is when you want to make the execution branch to a certain line, e.g. by a **GOTO** statement. You cannot use line numbers to specify the line in question. Instead, there is a feature called "line labels". The line you want to refer to must start with a line label, i.e. a number of characters appended by a colon (:). The line label must not start with a digit or interfere with any keyword (see chapter 4.8).

When you want to refer to a line marked with a line label, just enter the line label (without any colon), where you otherwise would have put the line number.

Finish the program by issuing an **IMMEDIATE ON** statement before you **RUN** it. The lines will automatically be numbered 10-20-30-40-50 etc., but the line numbers will not be visible before you **LIST** the program. Line labels will not be replaced by line numbers.

Two simple examples show the difference between using line numbers and line labels:

Line Numbers

```
10    GOSUB 1000
20    END
1000  SOUND 440,50
1010  RETURN

RUN

LIST
10    GOSUB 1000
20    END
1000  SOUND 440,50
1010  RETURN
```

Line Labels

```
IMMEDIATE OFF
GOSUB Q123
END
Q123:SOUND 440,50
RETURN
IMMEDIATE ON
RUN

LIST
10 GOSUB Q123
20 END
30 Q123: SOUND 440,50
40 RETURN
```

5.4 Programming Mode, cont'd.

*Warning! If there already is a program in the working memory, it will be deleted and cannot be restored unless it has been **SAVE**d.*

Programming Instructions

A number of instructions are used in connection with the editing of programs in the Programming Mode:

NEW

Before you enter the first program line, always issue a **NEW** statement in the Immediate Mode to **CLEAR** the printer's working memory, **CLOSE** all files and **CLEAR** all variables.

IMMEDIATE OFF

To write the program without entering line numbers manually, issue this statement in the Immediate Mode before the first line is entered.

REM (')

To make the program easier to understand, enter remarks and explanations on separate lines or in lines containing other instructions. Any characters preceded by **REM**, or its shorthand version **'** (single quotation mark), will not be regarded as part of the program and will not be executed. **REM** statements can also be used at the end of lines, if they are preceded by a colon (:).

END

Usually, subroutines are entered on lines with higher numbers than the main program. It is a good programming habit to finish the main program with an **END** statement in order to separate it from the subroutines. When an **END** statement is encountered, the execution is terminated and all **OPENed** files and devices are **CLOSEd**.

IMMEDIATE ON

If an **IMMEDIATE OFF** statement has been issued before starting to write the program, turn on the Immediate Mode again by means of an **IMMEDIATE ON** statement before starting the execution, i.e. a **RUN** statement is issued.

LIST

You can **LIST** the entire program, i.e. make the printer return the lines to the screen of the host. You can also choose to list part of the program or variables only. If you have edited the program without line numbers, the numbers automatically assigned to the lines at execution will now appear. **LIST** is usually issued in the Immediate Mode.

DELETE

Program lines can be removed using the **DELETE** statement in the Immediate Mode. Both single lines and ranges of lines in consecutive order can be deleted.

RENUM

The program lines can be renumbered, e.g. to provide space for new program lines, to change the order of execution, or to make it possible to **MERGE** to programs. Line references for **GOSUB**, **GOTO** and **RETURN** statements will be renumbered accordingly.

5.5 Conditional Instructions

Conditional instructions control the execution according to whether a numeric expression is true or false. UBI Fingerprint has one conditional instruction, which can be used in two different ways:

- **IF...THEN...[ELSE]**
- **IF...THEN...[ELSE]...ENDIF**

☞ ***TRUE and FALSE***
Also see:
• Chapter 4.9 (Relational Operators)

IF...THEN...[ELSE]

If a numeric expression is TRUE, then a certain statement should be executed, but if the numeric expression is FALSE, optionally another statement should be executed.

This example allows you to compare two values entered from the keyboard of the host.

```
10 INPUT "Enter first value ", A%
20 INPUT "Enter second value ", B%
30 C$="1:st value > 2:nd value"
40 D$="1:st value ≤ 2:nd value"
50 IF A%>B% THEN PRINT C$ ELSE PRINT D$
60 END
RUN
```

Another way to compare the two values in the example above is to use three IF...THEN statements:

```
10 INPUT "Enter first value ", A%
20 INPUT "Enter second value ", B%
30 C$="First value is larger than second value"
40 D$="First value is less than second value"
50 E$="First value and second value are equal"
60 IF A%>B% THEN PRINT C$
70 IF A%<B% THEN PRINT D$
80 IF A%=B% THEN PRINT E$
90 END
RUN
```

IF...THEN...[ELSE]...ENDIF

If is also possible to execute multiple **THEN** and **ELSE** statements. Each statement must be entered on a separate line and end of the instruction must be indicated by **ENDIF** on a separate line, as illustrated by the following example:

```
10 TIME$ = "121500":FORMAT TIME$ "HH:MM"
20 A%=VAL(TIME$)
30 IF A%>120000 THEN
40 PRINT "TIME IS ";TIME$("F"); ". ";
50 PRINT "GO TO LUNCH!"
60 ELSE
70 PRINT "CARRY ON - ";
80 PRINT "THERE'S MORE WORK TO DO!"
90 ENDIF
RUN
```

yields e.g.:

```
TIME IS 12:15. GO TO LUNCH!
```

5.6 Unconditional Branching

☞ ***Keyboard Control***
Also see:
• Chapter 15.1

GOTO

The most simple type of unconditional branching is the "waiting loop". This means that a program line branches the execution to itself, waiting for something to happen, for example a key being pressed or a communication buffer becoming full.

This example shows how the program waits for the key F1 to be pressed (line 30). Then a signal is emitted by the printer's buzzer:

```
10 ON KEY (10) GOSUB 1000
20 KEY (10) ON
30 GOTO 30
40 END
1000 SOUND 880,100
1010 END
RUN
```

It is also possible to branch to a different line. This is useful when you want create a waiting loop containing a number of lines.

Example:

```
10 INPUT "Enter a number:", A%
20 IF A%<0 THEN GOTO 100 ELSE GOTO 200
30 GOTO 10
40 END
100 PRINT "NEGATIVE VALUE"
110 GOTO 40
200 PRINT "POSITIVE VALUE"
210 GOTO 40
RUN
```

*The **GOTO** statement in line 30 diverts the execution back to line 10 over and over again until you type a value on the host (waiting loop). Depending on whether the value is less than 0 or not, the execution branches to one of two alternative lines (100 or 200), which print different messages to the screen. In both cases, the execution branches to line 40, where the program ends.*

There are more elegant ways to create such a program, but this example illustrates how **GOTO** always branches to a specific line. Line 20 is an example of conditional branching, which is explained in chapter 5.8.

5.7 Branching to Subroutines

GOSUB and RETURN

A subroutine is a number of program lines intended to perform a specific task, separately from the main program execution. Branching to subroutine can e.g. take place when:

- An error condition occurs.
- A condition is fulfilled, such as a certain key being pressed or a variable obtaining a certain value.
- A break instruction is received.
- Background communication is interrupted.

Another application of subroutines is branching to one and the same routine from different places in the same program. Thereby, you do not need to write the routine more than once and can make the program more compact.

The main instruction for branching to subroutines is the **GOSUB** statement. There are also a number of instructions for conditional branching to subroutines, which will be explained later in this chapter.

After branching, the subroutine will be executed line by line until a **RETURN** statement is encountered.

The same subroutine can be branched to as many times as you need from different lines in the main program. **GOSUB** remembers where the last branching took place, which makes it possible to return to the correct line in the main program after the subroutine has been executed. Subroutines may be nested, i.e. a subroutine may contain a **GOSUB** statement for branching to a secondary subroutine etc.

Subroutines should be placed on lines with higher numbers than the main program. The main program should be appended by an **END** statement to avoid unintentional execution of subroutines.

Example illustrating nested subroutines:

```
10    PRINT "This is the main program"
20    GOSUB 1000
30    PRINT "You're back in the main program"
40    END
1000  PRINT "This is subroutine 1"
1010  GOSUB 2000
1020  PRINT "You're back from subroutine 2 to 1"
1030  RETURN
2000  PRINT "This is subroutine 2"
2010  GOSUB 3000
2020  PRINT "You're back from subroutine 3 to 2"
2030  RETURN
3000  PRINT "This is subroutine 3"
3010  PRINT "You're leaving subroutine 3"
3020  RETURN
RUN
```

5.8 Conditional Branching

☞ ***Relational Operators***
Also see:
• Chapter 4.9

As the name implies, conditional branching means that the program execution branches to a certain line or subroutine when a specified condition is fulfilled. The following instructions are used for conditional branching:

IF...THEN GOTO...ELSE
If a specified condition is TRUE, the program branches to a certain line, but if the condition is FALSE, something else will be done.

Example:

```
10    INPUT "Enter a value: ",A%
20    INPUT "Enter another value: ",B%
30    IF A%=B% THEN GOTO 100 ELSE PRINT "NOT EQUAL"
40    END
100   PRINT "EQUAL"
110   GOTO 40
RUN
```

ON...GOSUB
Depending on the value of a numeric expression, the execution will branch to one of several subroutines. If the value is 1, the program will branch to the first subroutine in the instruction, if the value is 2 it will branch to the second subroutine and so on.

Example:

```
10    INPUT "Press key 1, 2, or 3 on host: ", A%
20    ON A% GOSUB 1000, 2000, 3000
30    END
1000  PRINT "You have pressed key 1": RETURN
2000  PRINT "You have pressed key 2": RETURN
3000  PRINT "You have pressed key 3": RETURN
RUN
```

ON...GOTO
This instruction is similar to **ON...GOSUB** but the program will branch to specified lines instead of subroutines. This implies that you cannot use **RETURN** statements to go back to the main program.

Example:

```
10    INPUT "Press key 1, 2, or 3 on host: ", A%
20    ON A% GOTO 1000, 2000, 3000
30    END
1000  PRINT "You have pressed key 1": GOTO 30
2000  PRINT "You have pressed key 2": GOTO 30
3000  PRINT "You have pressed key 3": GOTO 30
RUN
```

5.8 Conditional Branching, cont'd.

☞ ***Breaking the Execution***
Also see:
• Chapter 5.12

ON BREAK...GOSUB

When a **BREAK** condition occurs on a specified device, the execution will be interrupted and branched to a specified subroutine. There, you can e.g. let the printer emit a sound signal or display a message before the program is terminated. You can also let the program execution continue along a different path.

In this example the program is interrupted when the <Shift> and <Pause> keys on the printer's keyboard are pressed (default). The execution branches to a subroutine, which emits a siren-sounding signal three times. Then the execution returns to the main program, which is indicated by a long shrill signal. You can also issue a break interrupt by transmitting the character "#" (ASCII 35 dec.) from the host on the communication channel "uart1:".

```
10    BREAK 1,35
20    BREAK 1 ON
30    ON BREAK 0 GOSUB 1000:REM Break from keyboard
40    ON BREAK 1 GOSUB 1000:REM Break from host (#)
50    GOTO 40
60    SOUND 800,100
70    BREAK 1 OFF: END
1000  FOR A%=1 TO 3
1010  SOUND 440,50
1020  SOUND 349,50
1030  NEXT A%
1040  GOTO 60
RUN
```

ON COMSET...GOSUB

☞ ***Background Communication***
Also see:
• Chapter 7.8

When one of several specified conditions interrupts the background communication on a certain communication channel, the program branches to a subroutine, e.g. for reading the buffer. The interrupt conditions (end character, attention string and/or max. number of characters) are specified by a **COMSET** statement.

Example:

```
1     REM Exit program with #STOP&
10    COMSET1,"#","&","ZYX","=",50
20    ON COMSET 1 GOSUB 2000
30    COMSET 1 ON
40    IF A$ <> "STOP" THEN GOTO 40
50    COMSET 1 OFF
60    END
1000  END
2000  A$= COMBUF$(1)
2010  PRINT A$
2020  COMSET 1 ON
2030  RETURN
```

5.8 Conditional Branching, cont'd.

☞ ***Branching at Errors***
Also see:
• Chapter 16.3

Two instructions are used to branch to and from an error-handling subroutine when an error occurs:

ON ERROR GOTO
This statement branches the execution to a specified line when any kind of error occurs, ignoring the standard error-trapping routine. If line number is specified as 0, the standard error-trapping routine will be used.

RESUME
The **RESUME** statement is used to resume the program execution after an error-handling subroutine has been executed. **RESUME** is only used in connection with **ON ERROR GOTO** statements and can be used in five different ways:

RESUME	Execution is resumed at the statement where the error occurred.
RESUME 0	Same as **RESUME**.
RESUME NEXT	Execution is resumed at the statement immediately following the one that caused the error.
RESUME <ncon>	Execution is resumed at the specified line.
RESUME <line label>	Execution is resumed at the specified line label.

This example shows branching to a subroutine when an error has occurred. The subroutine determines the type of error and takes the appropriate action. In this example only one error; "1019 Invalid font" is checked. After the error is cleared by substituting the missing font, the execution will be resumed.

```
10    ON ERROR GOTO 1000
20    PRTXT "HELLO"
30    PRINTFEED
40    END
1000  IF ERR=1019 THEN FONT "Swiss 721 BT" ELSE GOTO 2000
1010  PRINT "Substitutes missing font"
1020  FOR A%=1 TO 3
1030  SOUND 440,50
1040  SOUND 359,50
1050  NEXT A%
1060  RESUME
2000  PRINT "Undefined error, execution terminated"
2010  END
RUN
```

5.8 Conditional Branching, cont'd.

☞ ***Keyboard Control and Key Id. No:s***
Also see:
• Chapter 15.1

ON KEY...GOSUB

All present UBI Fingerprint 7.xx-compatible printers are provided with a built-in keyboard. However, unless there is a program running in the printer, e.g. UBI Shell, the keys have no purpose (with the exception of <**Print**>, <**Feed**>, <**Shift**>, and <**Setup**> keys, which work in the Immediate Mode). To make use of the keyboard, each key must be enabled individually by means of a **KEY ON** statement and then be assigned to a subroutine using an **ON KEY GOSUB** statement. The subroutine should contain the instructions you want to be performed when the key is pressed.

In the statements **KEY (<id.>) ON**, **KEY (<id.>) OFF**, and **ON KEY (<id.>) GOSUB...**, the keys are specified by id. numbers enclosed by parentheses, see chapter 15.1.

Note that **ON KEY...GOSUB** excludes input from the printer's keyboard (see chapter 7.6) and vice versa.

*This example shows how the two unshifted keys <**F1**> (id. No. 10) and <**F2**> (id. No. 11) are used to change the printer's setup in regard of printout contrast.*

```
10    PRPOS 100,500
20    PRLINE 100,100
30    FONT "Swiss 721 BT"
40    PRPOS 100,300
50    MAG 4,4
60    PRTXT "SAMPLE"
70    KEY (10) ON : KEY (11) ON
80    ON KEY (10) GOSUB 1000
90    ON KEY (11) GOSUB 2000
100   GOTO 70
110   PRINTFEED
120   END
1000  SETUP "MEDIA,CONTRAST,-10%"
1010  PRPOS 100,100 : PRTXT "Weak Print"
1020  RETURN 110
2000  SETUP "MEDIA,CONTRAST,10%"
2010  PRPOS 100,100 : PRTXT "Dark Print"
2030  RETURN 110
RUN
```

5.9 Loops

GOTO
One type of loop has already been described in connection with the **GOTO** statement in chapter 5.6, where **GOTO** was used to refer to the same line or a previous line. There are also two more advanced type of loops:

FOR...NEXT
These statements are to used create loops, where a counter is incremented or decremented until a specified value is reached. The counter is defined by a **FOR** statement with the following syntax:

FOR<numeric variable>=<start value>TO<final value>[STEP<±interval>]

All program lines following the **FOR** statement will be executed until a **NEXT** statement is encountered. Then the counter will be updated according to the optional **STEP** value, or by the default value +1, and the loop will be executed again. This will be repeated until the final value, as specified by **TO <finalvalue>**, is reached. Then the loop is terminated and the execution proceeds from the statement following the **NEXT** statement.

FOR...NEXT loops can be nested, i.e. a loop can contain another loop etc. Each loop must have a unique counter designation in the form of a numeric variable. The **NEXT** statement will make the execution loop back to the most recent **FOR** statement. If you want to loop back to a different **FOR** statement, the corresponding **NEXT** statement must include the same counter designation as the **FOR** statement.

This example shows how five lines of text entered from the keyboard of the host can be printed with an even spacing:

```
10    FONT "Swiss 721 BT"
20    FOR Y%=220 TO 100 STEP -30
30    LINE INPUT "Type text: ";TEXT$
40    PRPOS 100, Y%
50    PRTXT TEXT$
60    NEXT
70    PRINTFEED
80    END
RUN
```

Here is an example of two nested FOR...NEXT loops:

```
10   FOR A%=20 TO 40 STEP 20
20    FOR B%=1 TO 2
30    PRINT A%,B%
40    NEXT : NEXT A%
RUN
```

Yields:

```
20    1
20    2
40    1
40    2
```

5.9 Loops, cont'd.

FOR...NEXT, cont'd.

This example shows how an incremental counter can be made:

```
10    INPUT "Start Value: ", A%
20    INPUT "Number of labels: ", B%
30    INPUT "Increment: ", C%
40    X%=B%*C%
50    FOR D%=1 TO X% STEP C%
60    FONT "Swiss 721 BT",24
70    PRPOS 100,200
80    PRTXT "TEST LABEL"
90    PRPOS 100,100
100   PRTXT "COUNTER: "; A%
110   PRINTFEED
120   A%=A%+C%
130   NEXT D%
RUN
```

WHILE...WEND

These statements are used to create loops where series of statements are executed provided a given condition is TRUE.

WHILE is supplemented by a numeric expression, that can be either TRUE (-1) or FALSE (0). If the condition is TRUE, all subsequent program lines will be executed until a **WEND** statement is encountered. The execution then loops back to the **WHILE** statement and the process is repeated, provided the **WHILE** condition still is TRUE. If the **WHILE** condition is FALSE, the execution bypasses the loop and resumes at the statement following the **WEND** statement.

☞ ***Relational Operators***
Also see:
• Chapter 4.9

WHILE...WEND statements can be nested. Each **WEND** statement matches the most recent **WHILE** statement.

*This example shows a program that keeps running in a loop (line 20–50) until you press the Y key on the host (ASCII 89 dec.), i.e. the **WHILE** condition becomes true.*

```
10    B%=0
20    WHILE B%<>89
30    INPUT "Want to exit? Press Y=Yes or N=No",A$
40    B%=ASC(A$)
50    WEND
60    PRINT "The answer is Yes"
70    PRINT "You will exit the program"
80    END
RUN
```

5.10 Program Structure

Although UBI Fingerprint gives the programmer a lot of freedom in how to compose his programs, based on experience we recommend that the structure below is more or less implemented, with the obvious exception of such facilities that are not needed.

❑ **Program Information**
- Program information, e.g. program type, version, release date and byline (**REM**).

❑ **Initiation**

Decides how printer will work and branch to subroutines.
- References to subroutines using e.g. **ON BREAK GOSUB**, **ON COMSET GOSUB**, **ON ERROR GOSUB**, **ON KEY GOSUB**.
- Printer setup using e.g. **SETUP**, **OPTIMIZE ON/OFF**, **LTS& ON/OFF**, **CUT ON/OFF**, **FORMAT DATE$**, **FORMAT TIME$**, **NAME DATE$**, **NAME WEEKDAY$**, **SYSVAR**.
- Character set and map tables (**NASC**, **NASCD**, **MAP**).
- Enabling keyboard (**KEY ON**, **KEYBEEP**, **KEYBMAP$**).
- Initial LED setting (**LED ON/OFF**).
- Open "console:" for output (**OPEN**)
- Assign string variables for each line in the display (**PRINT#**).
- Select current directory (**CHDIR**).
- Select standard I/O channel (**SETSTDIO**).
- Open communication channels (**OPEN**).
- Open files (**OPEN**).
- Define arrays (**DIM**).

❑ **Main Loop**

Executes the program and keeps it running in a loop.
- Reception of input data (**INPUT**, **INPUT#**, **INPUT$**, **LINE INPUT#**).
- Printing routine (**FORMFEED**, **PRINTFEED**, **CUT**).
- Looping instructions (**GOTO**).

❑ **Subroutines**
- Break subroutines (**BREAK ON/OFF**, **BREAK**).
- Background communication subroutines (**COM ERROR ON/OFF**, **COMSET**, **COMSET ON/OFF**, **COMBUF$**, **COMSTAT**).
- Subroutines for key-initiated actions.
- Subroutines for display messages.
- Error handling subroutines (**ERR**, **ERL**, **PRSTAT**).
- Label layouts subroutines.

5.11 Execution

To start the execution of the program currently residing in the printer's working memory, issue a **RUN** statement in the Immediate Mode, i.e. without a preceding line number. By default, the program will be executed in ascending line number order – with the exception of possible loops and branches – starting from the line with the lowest number, but you can optionally start the execution at a specified line.

You can also execute a program that is not **LOADed**.

If a program has been written without line numbers, the lines will be numbered 10-20-30-40-50.... etc.

The first program or hardware error that stops the execution will cause an error message to be returned to the screen of the host, provided there is a working two-way communication[1]. In case of program errors, the number of the line where the error occurred will also be reported by default, e.g. *"Field out of label in line 110"*. After the error has been corrected, the execution must be restarted by means of a new **RUN** statement, unless a routine for dealing with the error in question is included in the program.

☞ ***Standard Error-Handling***
Also see:
• Chapter 16.1

For demonstration purposes, we will now:
- write a short program without line numbers,
- execute it,
- and finally list it.

Note:
*For program instructions you can usually use upper- or lowercase characters at will, i.e. "**NEW**" and "**new**" will work the same way.*

```
NEW
Ok
IMMEDIATE OFF
Ok
REM This is a demonstration program
PRINT "This is the main program"
GOSUB sub1
END
sub1: PRINT "This is a subroutine":' Line label
RETURN
IMMEDIATE ON
Ok
RUN
```

yields:

```
This is the main program
This is a subroutine
Ok
LIST
```

yields:

```
10    REM This is a demonstration program
20    PRINT "This is the main program"
30    GOSUB SUB1
40    END
50    SUB1: PRINT "This is a subroutine" : ' Line label
60    RETURN
```

[1]/. For a working two-way communication, three conditions must be fulfilled:
• Serial communication
• Std IN channel = Std OUT channel
• Verbosity on

5.12 Breaking the Execution

In chapter 2 “Getting Started” at the beginning of this manual, the methods of breaking a startup program was briefly explained. Startup programs (autoexec files) start up automatically when the printer is turned on and continues to run infinitely by means of some kind of loop.

You can – by default – break a program by pressing the <**Shift**> key and keep it pressed while you also press down the <**Pause**> key. There is – by default – no break facilities from the host via any communication channel. Therefore, it is strongly recommended always to include break facilities in startup programs.

If the startup program resides in a memory card, you can of course turn off the printer, remove the card and start up again.

Four instructions can be used for providing a program with a break interrupt facility:

BREAK	Specifies an interrupt character.
BREAK...ON	Enables break interrupt.
BREAK...OFF	Disables break interrupt.
ON BREAK...GOSUB...	Branches the execution to a subroutine when a break interrupt is executed.

In all break-related instructions, the serial communication channels[1] and the keyboard are referred to by numbers:
0 = "console:" (i.e. the printer's keyboard)
1 = "uart1:"
2 = "uart2:"

BREAK

The **BREAK** statement specifies an interrupt character by its decimal ASCII value. **BREAK** can be separately specified for each **serial** communication channel and for the printer's built-in keyboard.

The interrupt character for all serial channels is by default ASCII 03 dec. (ETX). Also see **BREAK...ON**.

The interrupt character from the printer's keyboard is by default ASCII 158 dec. (<**Shift**> + <**Pause**> keys). Also see **BREAK...ON**.

BREAK...ON

Break interrupt for all serial communication channels is **disabled** by default, but can be enabled by means of a **BREAK...ON** statement for the channel in question.

Break interrupt from the keyboard is **enabled** by default.

BREAK... OFF

The **BREAK...OFF** statement revokes **BREAK...ON** for the specified device and deletes the specified break character from the printer's memory.

[1]/. BREAK does not work on the parallel Centronics channel.

5.12 Breaking Execution, cont'd.

Note:
A break interrupt character is saved in the printer's temporary memory, and will not be removed before the printer is restarted, unless you specifically delete it by a ***BREAK...OFF*** *statement for the device in question.*

ON BREAK ...GOSUB...
This instruction is not necessary for issuing a break interrupt, but is useful for making the printer perform a certain task when a break occurs, e.g. branch the execution to another part of the program, show a message in the display, emit a warning signal, ask for a password etc. **ON BREAK... GOSUB...** can be specified separately for each serial communication channel and for the keyboard.

This example shows how a break interrupt will occur when you press the X-key (ASCII 88 dec.) on the host connected to "uart1:". A signal is emitted and a message appears in the printer's display.

```
10    BREAK 1,88
20    BREAK 1 ON
30    OPEN "console:" FOR OUTPUT AS 1
40    PRINT #1 : PRINT #1
50    PRINT #1, "Press X"
60    PRINT #1, "to break program";
70    ON BREAK 1 GOSUB 1000
80    GOTO 80
90    BREAK 1 OFF
100   END
1000  SOUND 880,50
1010  PRINT #1 : PRINT #1
1020  PRINT #1, "PROGRAM"
1030  PRINT #1, "INTERRUPTED";
1040  RETURN 90
RUN
```

5.13 Saving the Program

Saving in Printer
When you are satisfied with the program, you can **SAVE** it in the printer's permanent memory ("c:"), in the printer's temporary memory ("tmp:") or in a DOS-formatted memory card ("card1:"), see chapter 6.1. Obviously, if you save it in "tmp:", it will be lost at power off or at a power failure.

It is also recommended to **LIST** the program back to the host and make backup copy, e.g. on a floppy disk.

Naming the Program
When you save a program for the first time, you must give it a name consisting of up to 30 uppercase characters including possible extension. If you omit the extension, the firmware will add the extension ".PRG" automatically. When naming the program, consider conventions and restrictions imposed by the operating system of the host, e.g. MS-DOS, Windows 3.11, Windows 95 etc.

If the program or file name starts with a period character, it will be regarded a system file, see **FILES** and **FORMAT** statements in the UBI Fingerprint 7.xx Reference Manual.

5.13 Saving the Program, cont'd.

The following names are used for standard *UBI Fingerprint* programs and should not be used:

- **`.setup.saved`** *Current setup values*
- **`.theDefaultSetup`** *Default setup values*
- **`.ubipfr1.bin`** *Standard fonts*
- **`APPLICATION`** *UBI Shell auxiliary file*
- **`DIRECT`** *UBI Shell auxiliary file*
- **`ERRHAND.PRG`** *Error Handler*
- **`FILELIST.DAT`** *UBI Shell auxiliary file*
- **`FILELIST.PRG`** *List the lines of a file*
- **`LBLSHTXT.PRG`** *UBI Shell auxiliary file*
- **`LINE_AXP.PRG`** *UBI Shell Line Analyzer*
- **`LSHOPXP1.SUB`** *UBI Shell auxiliary file*
- **`MKAUTO.PRG`** *Create a startup (autoexec) file*
- **`PUP.BAT`** *UBI Shell Startup file*
- **`SHELLXP.PRG`** *UBI Shell startup program*
- **`STDIO`** *UBI Shell auxiliary file*
- **`WINXP.PRG`** *UBI Shell auxiliary file*

Examples:

`SAVE "PROGRAM1"`

saves the program as PROGRAM1.PRG in the current directory (by default "c:").

☞ ***Current Directory***
Also see:
• Chapter 6.1

`SAVE "card1:PROGRAM1.TXT"`

saves the program as PROGRAM1.TXT in a DOS-formatted memory card inserted in the printer's optional memory card adapter.

Protecting the Program

When a program is **SAVEd**, it can optionally be protected, i.e. it cannot be listed after being loaded and program lines cannot be changed, added or deleted. Once a program has been protected, it cannot be deprotected. Thus, make an unprotected backup copy as a safety measure, should you need to make any changes later.

Example (saves and protects the program as PROGRAM1.PRG in the current directory (by default "c:"):

`SAVE "PROGRAM1.PRG",P`

Saving Without Line Numbers

A program can also be **SAVEd** without line numbers to make it easier to **MERGE** it with another program without risking that the line numbers interfere. Both programs should make use of line labels for referring to other lines, e.g. in connection with loops and branching instructions.

Example (saves the program as PROGRAM1.PRG without line numbers in the current directory (by default "c:"):

`SAVE "PROGRAM1.PRG",L`

5.13 Saving the Program, cont'd.

Making Changes

If you **LOAD** a program, possibly make some changes and then **SAVE** the program under the original name and in the original directory, the original program will be replaced.

Example (changes the value of a variable in line 50 of a program and replaces the original version with the changed version):

```
LOAD "PROGRAM1.PRG"
50 A%=300
SAVE "PROGRAM1.PRG"
```

Making a Copy

The easiest way to copy a program is to use a **COPY** statement. You can optionally include directory references in the statement.

Example (copies a program from the permanent memory to a DOS-formatted memory card and gives the copy a new name):

```
COPY "c:FILELIST.PRG","card1:COPYTEST.PRG"
```

If you **LOAD** a program and then **SAVE** it under a new name and/or in another directory, you will create a copy of the original program.

Example (creates a copy of the program LABEL1.PRG and gives the copy the name LABEL2.PRG):

```
LOAD "LABEL1.PRG"
SAVE "LABEL2.PRG"
```

Renaming a Program

To rename a program, **LOAD** it, **SAVE** it under a new name, and finally **KILL** the original program.

Example (renames LABEL1.PRG with the name LABEL2.PRG):

```
LOAD "LABEL1.PRG"
SAVE "LABEL2.PRG"
KILL "LABEL1.PRG"
```

Note: The same general principles also apply to files!

Saving in Non DOS-formatted Memory Cards

Saving a program or file in non DOS-formatted memory cards requires special equipment such as a PROM programmer and the aid of the UBI Configuration program, which is included in UBI Toolbox. You can edit and test the program in the printer's working memory as described earlier in this chapter. When it works properly, **LIST** it back to the host computer or **COPY** it to a serial communication channel. Save the file in the host and use UBI Configuration to convert it to a format suitable for the memory card programming device. Refer to the UBI Toolbox Programmer's Manual for further information.

5.13 Saving the Program, cont'd.

Creating a Startup Program

The MKAUTO.PRG program is used to create so called startup programs or autoexec-files, i.e. programs that will be **LOADed** and **RUN** automatically as soon as the power to the printer is turned on. Usually, a startup program contains some kind of loop which makes it run infinitely, awaiting some input or action from the operator.

There must not be more than one startup program in each part of the memory, i.e.:

- **DOS-formatted memory cards ("card1:"):**
  Max. one startup program per card.
- **Non DOS-formatted memory cards ("rom:"):**
  Max. one startup program per card.
- **Printer's permanent memory ("c:"):**
  Max. one startup program.

If there are more than one startup file in the printer's memory, they will be used with the following priority:

1. An autoexec.bat file in any type of memory card ("rom:" or "card1:") that was inserted at start-up.
2. An autoexec.bat file in printer's permanent memory ("c:")
3. The pup.bat file (UBI Shell) in the systems part of the printer's permanent memory ("rom:")

The MKAUTO.PRG program is included in the systems part of the printer's memory ("rom:") and consists of the following lines:

```
10    OPEN "AUTOEXEC.BAT" FOR OUTPUT AS 1
20    INPUT "Startup file name:",S$
30    PRINT#1,"RUN";CHR$(34);S$;CHR$(34)
40    CLOSE1
```

A startup program can easily be created from an ordinary program using the following method:

- After having written and tested the program, **SAVE** it.
- Enter the following statement:
  `RUN "rom:MKAUTO"`
- The following prompt will be displayed on the screen:
  `STARTUP FILE NAME?`
- Type the name of the program you just **SAVEd** (with or without the extension .PRG) and press the Carriage Return key.
- **Ok** on the screen indicates that the operation is ready.
- The startup program will be stored in the printer's current directory (by default "c:", i.e. the printer's permanent memory).
- When you restart the printer, the new startup program will start running, provided there is no other startup program with higher priority (see previous page).

☞ *Current Directory*
Also see:
• Chapter 6.1

To undo the operation, use the statement:

```
KILL "AUTOEXEC.BAT"
```

This will not erase the original program, but it will no longer be used as a startup program. Note that you cannot **KILL** startup programs stored in "rom:".

5.14 Rebooting the Printer

Rebooting the printer has the same consequences as turning off and then on the power.

REBOOT
This statement allows you to reboot the printer from the host or as a part of the program execution.

When the printer is rebooted, or the power to the printer is turned on, a number of things happens:

- The printer's temporary memory ("tmp:") is erased, i.e. any program not already **SAVEd** to "c:" or "card1:" will be irrevocably lost, all buffers will be emptied, all files will be closed, all date- and time-related formats will be lost, all arrays will be lost and all variables will be set to zero. Fonts and images stored in the temporary memory will be erased.
- All parameters in UBI Fingerprint instructions will be reset to default.
- The printer performs a number of self-diagnostic tests, e.g. printhead resistance check (certain models only) and memory checksum calculations.
- The printer checks for possible optional devices like interface boards or cutter.
- The various parts of the printer's memory are searched for possible startup programs as described in chapter 5.13. The first startup program encountered will be executed.

Note that rebooting does not change the printer's setup, unless any physical changes has been done to the printer during the power-off period, such as a change of printhead or installation or removal of an interface board.

6. File System

6.1 Printer's Memory

Abbreviations:
SIMM = Standard In-line Memory Module
DRAM = Dynamic Random Access Memory
VFM = Virtual Small Block File Manager
FOS = File Operating System
ROM = Read Only Memory

[1]/. This applies to the following instructions:
???????
???????

Note:
To provide compatibility with earlier versions of UBI Fingerprint, the device "ram:" is equal to "c:".

The printer's memory consists of a number of parts, or directories:

- **Permanent Memory ("rom:" and "c:")**
  The permanent memory resides in one or several flash memory SIMMs. A flash SIMM will keep it contents when the power to the printer is off without the aid of a battery backup system.

  Each SIMM contains a number of sectors. Some sectors that are both read and write capable, whereas other are read-only. A read/write sector is typically 256 kbytes and consists a number of pointers and blocks, each with a size of 1 kbyte. A pointer can refer to several blocks and also to another pointer. If just a single character (one byte) is entered[1], a whole 1 kbyte block and one 1 kbyte pointer will be occupied. On the other hand, e.g. 4.5 kbytes of data requires one 1 kbyte pointer and five 1 kbyte blocks.

  When there are no free blocks left in any sector and at power up, the memory will automatically be reorganized to save space. Before reorganization, the sector is copied to a temporary sector for safety reason if something should go wrong. Files are rewritten into as few blocks as possible and the number of pointers is thus reduced. Then the sector is erased and the content is copied back from the temporary sector. This takes some time and makes the flash memory comparatively slow.

  One SIMM must always be present and contains a boot sector and a number of sectors containing the so called “kernel”.

  There is also a temporary area for paper feed info and odometer values. All these sectors are read-only and are included in the device **"rom:"**.

  The remaining part of the same flash memory SIMM contains a number of read/write sectors and is designated as device **"c:"**. If there are additional flash SIMMs for the permanent memory, they are also included in the device **"c:"**.

  The following table illustrates the boot flash SIMM for an EasyCoder 501XP/601 XP printer:

Device	Size	Sector	Used for
c:	256 kbyte	Intel VFM FOS	Customer's programs, files, images etc.
	256 kbyte	Intel VFM FOS	
	256 kbyte	TMP area (FOS)	
rom:	256 kbyte	Kernel	UBI Fingerprint firmware, bar codes, standard fonts, standard images, UBI Shell, auxiliary programs[1], setup values
	256 kbyte	Kernel	
	256 kbyte	Kernel	
	256 kbyte	Kernel	
	192 kbyte	Kernel	
rom:	16 kbyte	TMP area	Paper feed info, odometer value
	16 kbyte	Parameters	
rom:	32 kbyte	Boot	Startup

6.1 Printer's Memory, cont'd.

- **Temporary Memory ("temp:")**
  The temporary memory (device "tmp:") is a read/write DRAM (Dynamic Random Access Memory) and residies in one or several SIMM packages. It has no backup and will be completely erased at power-off.

  The temporary memory is used for the following purposes:
  - To execute UBI Fingerprint instructions. At startup the kernel in the permanent memory is copied to the temporary memory, where all UBI Fingerprint instructions are executed and the print image bitmaps are created.
  - To be used for the print image buffers.
  - To be used for the font cache.
  - To be used for the Receive/Transmit buffers. Each serial communication channel must have one buffer of each kind. The size of each buffer is decided separately by the setup.
  – To be used for communication buffers. In a program, you may set up one communication buffer for each communication channel. This makes it possible to receive data simultaneously from several sources to be fetched at the appropriate moment during the execution of the program.
  - To store data that do not need to be saved after power-off.
  - To temporarily store data before they are copied to the permanent memory or to a memory card.

  The latter purpose is important considering how the permanent memory works. Since the permanent flash memory is comparatively slow, in connection with certain instructions (see previous page) it is more efficient to create files in the temporary memory and then save them to the permanent memory. When speed is important, also avoid saving data that nevertheless will be of no use after power off in the temporary memory.

  Note that there is no fixed division of the temporary memory. After the firmware has been copied to it and the Receive/Transmit buffers have been set according to the setup, the remaining memory will be shared between the various tasks.

- **DOS-Formatted Memory Cards: ("card1:")**
  The built-in memory can be supplemented with a DOS-formatted memory card that is inserted in the printer's memory card adapter. Such a card is referred to as "card1:" and can be both read from and written to. In order to retain its content when the power to the printer is off, each SRAM memory card is fitted with an internal battery.

6.1 Printer's Memory, cont'd.

- **Non DOS-Formatted Memory Cards ("rom:")**
  A non DOS-formatted, preprogrammed memory card can be inserted to supplement or update the built-in ROM memory. There are three types of ROM cards:
  - ***Font Cards*** are used to supplement the standard fonts in the permanent memory.
  - ***Font Install Cards*** are used to install addition fonts in the printer's permanent memory.
  - ***Firmware Cards*** are used to intall a new firmware version (kernel) in the printer's permanent memory.

- **Other Memory Devices ("storage:")**
  The "storage:" device is a small and slow memory device that is used for special application. It should not be used for normal UBI Fingerprint preogramming.

Current Directory

"Current directory" means the directory the UBI Fingerprint software will use unless you specifically instruct it to use another directory. By default, the current directory is "c:".

To appoint another directory as current directory, use a **CHDIR** statement.

Example:
Changing directory from the default directory ("c:") to "tmp:" and back.

```
10 CHDIR "tmp:"
. . . . .
90 CHDIR "c:"
```

Checking Free Memory

You can check the size of the memory in the current directory and see how much free space there is by issuing a **FILES** statement in the immediate mode.

Another way is to use the **FRE** function to make a small instruction, that returns the number of free bytes in the printer's temporary memory, for example:

```
PRINT FRE(1)
```

yields e.g.:

```
391248
```

Providing More Free Memory

In order to free more memory space in the temporary memory, you can use a **CLEAR** statement to empty all strings, set all variables to zero and reset all arrays to default. If even more memory is required, you will have to consider either to **KILL** some programs or files, or to use **REMOVE IMAGE** to delete some images stored in "c:" and/or "tmp:". If the printer is not fitted with the maximum size memory, you could also fit more or larger Flash or DRAM SIMM packages after having made backup copies on the host.

6.1 Printer's Memory, cont'd.

Formatting the Permanent Memory

The printer's permanent memory ("c:") can be formatted either partially or completely.

`FORMAT "c:",A`
erases all files in the device "c:" (hard formatting).

`FORMAT "c:"`
erases all files except those starting with a period (.) character (soft formatting). System files are provided with such a period character, e.g. .ubifr1.bin.

Formatting SRAM Memory Cards

An SRAM-type memory card, inserted in the printer's memory card adapter, can be formatted to MS-DOS format by means of a **`FORMAT`** statement, e.g.:

```
FORMAT "card1:",208,512,A
```

6.2 Files

☞ *Fonts, Bar Codes and Images*
Also see:
• Chapter 12 (Fonts)
• Chapter 13 (Images)
• Chapter 14 (Bar codes)

File Types

A number of different types of files can be stored in the various parts of the printer's memory. They can be divided into four main groups:

- Program Files
- Data Files
- Image Files
- Font Files

Object files, fonts, bar codes and images are not treated as files by the UBI Fingerprint firmware.

File Names

The name of a file may consist of up to 30 characters including extension, but possible restrictions imposed by the operating system of the host should be considered if the file is to be transferred. Refer to chapter 5.13 for a list of reserved file names.

☞ *Current Directory*
Also see:
• Chapter 6.1

Listing Files

The files stored in the printer's memory can be listed by means of a **`FILES`** statement. By default, the files stored in the current directory will be listed. Optionally, another directory can be selected by adding a device reference to the **`FILES`** statement, e.g.:

`FILES`	lists all files in the **current** directory.
`FILES "c:"`	lists all files in the read/write part of the permanent memory
`FILES "rom:"`	lists all files stored in the read-only part of the permanent memory (kernel) and in any inserted non DOS-formatted memory card.
`FILES "tmp:"`	lists all files stored in the printer's temporary memory.
`FILES "card1:"`	lists all files stored in any inserted DOS-formatted memory card.

6.2 Files, cont'd.

> ☞ ***Standard OUT channel***
> Also see:
> • Chapter 7.1

You can **COPY** a file to the standard OUT channel, where it will be printed on the screen of the host, e.g.:

```
COPY "[device]filename", "uart1:"
```

The FILELIST.PRG program included in the UBI Fingerprint firmware is used to **LIST** a line-orientated file to the standard OUT channel:

- On your terminal, enter:
  `RUN "rom:FILELIST.PRG"`
- The printer will respond by prompting you to enter the name of the file to be listed:
  `Filename?`
- Enter the filename, possibly preceded by a directory reference, e.g.
  `"c:*.*"`

6.3 Program Files

Program File Types

Program files are used to run and control the printer and to produce labels or other printouts. A program file is always composed of numbered lines, although the numbers may be invisible during the editing process (see chapter 5.4).

A special case of program files is startup files, i.e. files that automatically start running when the printer is turned on (also called “autoexec-files”). Startup files were explained in chapter 5.13 “Creating a Startup Program”.

> ☞ ***Creating, Saving, Copying, Killing and Executing Program Files***
> Also see:
> • Chapter 5.11 and 5.13

> ☞ ***Standard OUT Channel***
> Also see:
> • Chapter 7.1

Instructions

The following instructions are used for creating and handling program files:

LOAD	Copies a specified program file to the printer's working memory.
LIST	Lists the program file in the working memory to the standard OUT channel, usually the screen of the host.
MERGE	Adds copy of a specified program file to the program file currently residing in the printer's working memory.
RUN	Executes the instruction in the program file. Must be issued in the Immediate Mode, i.e. not in a numbered line.

> ☞ ***Current Directory***
> Also see:
> • Chapter 6.1

SAVE	Saves a copy of the program file in the current directory or, optionally, in another specified directory. If a file with the same name already exists the that directory, it will be replaced by the new file.
NEW	Clears the working memory to allow a new program file to be created.
COPY	Copies a file to another name and/or directory.
KILL	Deletes a file from the printer's permanent memory ("c:"), the printer's temporary memory ("tmp:") or from a DOS-formatted memory card ("card1:").

6.4 Data Files

Data File Types

Data files are used by the program files for storing various types of data and can be divided into several subcategories:

- Sequential Input Files — *See chapter 7.4*
- Sequential Output Files — *See chapter 8.3*
- Sequential Append Files — *See chapter 8.3*
- Random Access Files — *See chapters 7.5 and 8.4*

Instructions

The following instructions are used in connection with the creation and handling of data files:

OPEN	Creates and/or opens a file for a specified mode of access and optionally specifies the record size in bytes.
CLOSE	Closes an **OPENed** file.
REDIRECT OUT	Creates a file to which the output data will be redirected (see chapter 8.2).
TRANSFERSET	Sets up the transfer of data between two files.
TRANSFER$	Executes the transfer of data between two files according to **TRANSFERSET**.
COPY	Copies a file to another name and/or directory.
KILL	Deletes a file.
LOC	Returns the position in an **OPENed** file.
LOF	Returns the length in bytes of an **OPENed** file.

6.5 Image Files

☞ ***Images***
Also see:
• Chapter 14

Image files in .PCX format can be downloaded and installed in the printer's memory by means of the statement **IMAGE LOAD**.

Image files in .PCX format that have been downloaded to the printer's memory using Kermit file transfer protocol (see chapter 6.8) or stored in a DOS-formatted memory card cannot be used to produce a printable image before they have been converted to UBI Fingerprint's internal bitmap format by means of the following instruction:

```
RUN "pcx2bmp <name of .PCX file> <name of image>"
```

Image files in Intelhex format, or the formats UBI00, UBI01, UBI02, UBI03 or UBI10, can be downloaded and converted to images using the **STORE IMAGE** and **STORE INPUT** statements.

Images files can be listed by means of **FILES** statements.

6.6 Font Files

☞ ***Single and Double-byte Fonts***
Also see:
• Chapter ??

Font files are files in TrueDoc (*.PFR) or TrueType (*.TTF) format and contain scalable single or double-byte fonts complying with the Unicode standard. The printer's standard complement of single-byte fonts can be supplemented with additional fonts by downloading font files to the printer using Kermit file transfer protocol (see **TRANSFER KERMIT** in chapter 6.8) or using an **IMAGE LOAD** statement. After a font file has been downloaded, the corresponding font can be used immediately without any need for a reboot.

☞ ***Font Install Card and Font Card***
Also see:
• Chapter ??

Additional fonts can also be installed using a Font Install Card or be read from a Font Card. Note that since most double-byte fonts are very large, there may not be enough memory space in the printer to accommodate such fonts. In such a case, use a Font Card.

Font files can be listed by means of a **FILES** statements.

6.7 Transferring Text Files

Text files, e.g. program files and data files in ASCII format, can be downloaded via a communication program in the host, e.g. Windows Terminal ("Transfers; Send Text File").

Text files can be transferred back to the host, e.g. for backup purposes, by **LOADing** the file and **LISTing** it to a communication program in the host.

6.8 Transferring Binary Files using Kermit

Font files and some image files come in binary format and can be downloaded from the host to the printer or vice versa using the Kermit file transfer protocol, which is commonly used for binary transfer of data and is included in many communication programs, e.g. DCA Crosstalk, MS Windows Terminal, and MS Works.

Warning!
Tests have shown that MS Windows Terminal versions 3.0 and 3.1 are unable to receive a file from the printer, even if they are capable of sending a file to the printer.

More information on the Kermit protocol can be found in the manual of the communication program or in the reference volume "Kermit – A File Transfer Protocol" by Frank da Cruz (Digital Press 1987, ISBN 0-932376-88-6).

TRANSFER KERMIT

☞ ***Standard IN and OUT Channels***
Also see:
• Chapter 7.1

The **TRANSFER KERMIT** statement allows you to specify direction (Send or Receive), file name, input device and output device. By default, a file name designated "KERMIT.FILE" will be transferred on the standard IN or OUT channel.

Note that there is a 30 sec. timeout between the issuing of the ***TRANSFER KERMIT "R"*** *statement and the start of the transmission.*

Example:
The printer is set up to receive a file on the standard IN channel.

```
TRANSFER KERMIT "R"
```

6.8 Transferring Binary Files using Kermit, cont'd.

☞ ***Arrays***
Also see:
• Chapter 6.10

TRANSFER STATUS
After a file have been transferred by means of a **TRANSFER KERMIT** statement, the transfer can be checked using the **TRANSFER STATUS** statement. The statement will place the result of the check into two one-dimensional arrays:

5-element numeric array (requires a DIM stmt):

Element 0 returns:	Number of packets
Element 1 returns:	Number of NAKs
Element 2 returns:	ASCII value of last character
Element 3 returns:	Last error
Element 4 returns:	Block check type used

2-element string array (requires no DIM stmt):

Element 0 returns:	Type of protocol, i.e. "KERMIT"
Element 1 returns:	Last file name received

Example:

```
10    TRANSFER KERMIT "R"
20    DIM A%(4)
30    TRANSFER STATUS A%,B$
40    PRINT A%(0), A%(1), A%(2), A%(4), A%(4)
50    PRINT B$(0), B$(1)
RUN
```

If you want to transfer a file from one printer to another printer, start by transferring the file to the host. Then disconnect the first printer and download the file to the second printer (or have the two printers connected to separate serial ports). After the transfer of programs between two connected printers is completed, you can check if the transfer was successful by means of a **CHECKSUM** function.

6.9 Transferring Files Between Printers

Note:
Do not confuse CHECKSUM with CSUM, see chapter 6.10 "Arrays".

CHECKSUM
The **CHECKSUM** function uses an advanced algorithm on parts of the printer's internal code. Thus, calculate the **CHECKSUM** on the program in the transmitting printer before the transfer. After the transfer is completed, **LOAD** the program in the receiving printer and perform the same calculation. If the checksums are identical, the transfer was successful.

*Note that the algorithm was changed in UBI Fingerprint 4.0. Thus, the **CHECKSUM** function will return other checksums in printers using earlier versions of UBI Fingerprint than 4.0 compared to printers using 4.0 or later versions. If possible, use the same UBI Fingerprint version in both printers.*

Example:
This example calculates the checksum in the lines 10–90000 in the program "DEMO.PRG".

```
LOAD "DEMO.PRG"
PRINT CHECKSUM (10,90000)
```

6.10 Arrays

Variables containing related data may be organized in arrays. Each value in an array is called an element. The position of each element is specified by a subscript, one for each dimension (max 10). Each array variable consists of a name and a number of subscripts separated by commas and enclosed by parentheses, for example **ARRAY$(3,3,3)**.

The number of subscripts in an array variable, the first time (regardless of line number) it is referred to, decides its number of dimensions. The number of elements in each dimension is by default restricted to four (No. 0 – 3).

Four instructions are specifically used in connection with arrays:

DIM	Specifies the size of an array in regard of elements and dimensions.
SORT	Sorts the elements in a one-dimensional array in ascending or descending order.
SPLIT	Splits a string into an array.
CSUM	Returns the checksum for a string array.

DIM

If more than 4 elements are needed, or if you want to limit the size of the array, a **DIM** statement can be used to specify the size of the array in regard of the number of dimensions as well as the number of elements in each dimension. In most cases, one- or two-dimensional arrays will suffice.

This example shows how three 1-dimensional, 5-element arrays can be used to return 125 possible combinations of text strings:

```
10   DIM TYPE$(4),COLOUR$(4),SIZE$(4)
20   TYPE$(0)="SHIRT"
30   TYPE$(1)="BLOUSE"
40   TYPE$(2)="TROUSERS"
50   TYPE$(3)="SKIRT"
60   TYPE$(4)="JACKET"
70   COLOUR$(0)="RED"
80   COLOUR$(1)="GREEN"
90   COLOUR$(2)="BLUE"
100  COLOUR$(3)="RED"
110  COLOUR$(4)="WHITE"
120  SIZE$(0)="EXTRA SMALL"
130  SIZE$(1)="SMALL"
140  SIZE$(2)="MEDIUM"
150  SIZE$(3)="LARGE"
160  SIZE$(4)="EXTRA LARGE"
170  INPUT"Select Type (0-4): ", A%
180  INPUT"Select Colour (0-4): ", B%
190  INPUT"Select Size (0-4): ", C%
200  PRINT TYPE$(A%)+", "+COLOUR$(B%)+", "+SIZE$(C%)
RUN
```

6.10 Arrays, cont'd.

SORT

The **SORT** statement is used to sort a one-dimensional array in ascending or descending order according the character's ASCII values in the Roman 8 character set. You can also choose between sorting the complete array or a specified interval. For string arrays, you can select by which character position the sorting will be performed.

This example shows how one numeric array is sorted in ascending order and one string array is sorted in descending order according to the fifth character in each element:

```
10    FOR Q%=0 TO 3
20    A$=STR$(Q%)
30    ARRAY%(Q%)=1000+Q%:ARRAY$(Q%)="No. "+A$
40    NEXT Q%
50    SORT ARRAY%,0,3,1
60    SORT ARRAY$,0,3,-5
70    FOR I%=0 TO 3
80    PRINT ARRAY%(I%), ARRAY$(I%)
90    NEXT I%
RUN
```

yields:

```
1000  No. 3
1001  No. 2
1002  No. 1
1003  No. 0
```

SPLIT

The **SPLIT** function is used to split a string expression into elements in an array and to return the number of elements. A specified character indicates where the string will be split.

In this example a string expression is divided into six parts by the separator character "/" (ASCII 47 dec.) and arranged in a six element array:

```
10    A$="ONE/TWO/THREE/FOUR/FIVE/SIX"
20    X$="ARRAY$"
30    DIM ARRAY$(6)
40    B%=SPLIT(A$,X$,47)
50    FOR C%=0 TO (B%-1)
60    PRINT ARRAY$(C%)
70    NEXT
RUN
```

yields:

```
ONE
TWO
THREE
FOUR
FIVE
SIX
```

6.10 Arrays, cont'd.

Note!
Do not confuse CSUM with CHECKSUM, see chapter 6.9.

CSUM

The checksum for string arrays can be calculated according to one of two different algorithms (LRC or DRC) and returned by means of the **CSUM** statement.

In this example, the checksum of a string array is calculated according both to the LRC (Logitudinal Redundancy Check) and the DRC (Diagonal Redundancy Check) algorithms:

```
10    FOR Q%=0 TO 3
20    A$=STR$(Q%)
30    ARRAY$(Q%)="Element No. "+A$
40    NEXT
50    CSUM 1,ARRAY$,B%:PRINT "LRC checksum: ";B%
60    CSUM 2,ARRAY$,C%:PRINT "DRC checksum: ";C%
RUN
```

yields:

```
LRC checksum: 0
DRC checksum: 197
```

7. Input to UBI Fingerprint

7.1 Standard I/O Channel

☞ *Output from UBI Fingerprint*
See:
• Chapter 8

The standard IN and standard OUT channels are the default channels for input to the printer or output from the printer respectively (in both cases "uart1:" by default). In most instructions, you can override the standard IN or OUT channel by specifying another channel. Usually, the same channel is used for both input and output, but different channels can be specified.

SETSTDIO
You can appoint any of the following communication channels as standard IN and/or standard OUT channel by means of the **SETSTDIO** statement:

Standard IN channel	*Standard OUT channel*
0 = "console:"[1]	0 = "console:"[1]
1 = "uart1:" (default)	1 = "uart1:" (default)
2 = "uart2:"	2 = "uart2:"
4 = "centronics:"[2]	

[1]/. Do not select "console:" as both std in and out channel, since it would only make characters entered on the printer's key-board appear in the display.

[2]/. The parallel communication channel "centronics:" can only be used for input (one-way communication only).

7.2 Input from Host (Std IN Channel only)

The std IN channel is used for sending instructions and data from the host to the printer in order to control the printer in the immediate mode, to write programs in the programming mode, to download program files and to transmit input data.

Some instructions receives data on the std IN channel only:

INKEY$	Reads the 1:st character in the receive buffer.
INPUT	Receives input data during execution of a program.
LINE INPUT	Assigns an entire line to a string variable.

7.3 Input from Host (Any Channel)

The following instructions are used to receive input from **any** communication channel (incl. the std IN channel). The same instructions are also used to read sequential files, see chapter 7.4:

OPEN	Opens a channel for sequential INPUT.
INPUT#	Receives input data during execution of a program on the specified channel.
INPUT$	Reads a string of data from the specified channel.
LINE INPUT#	Assigns an entire line from the specified channel to a string variable.
CLOSE	Closes the channel.

7.4 Input from a Sequential File

Refer to chapter 7.3 for a summary of instructions used for reading sequential files.

OPEN
Before any data can be read from a sequential file (or a communication channel other than the std IN channel), it must be **OPENed** for **INPUT** and assigned a number, which is used when referred to in other instructions. The number mark (**#**) is optional. Up to 10 files and devices can be open at the same time.

Example: The file "ADDRESSES" is opened for input as number 1:

```
OPEN "ADDRESSES" FOR INPUT AS #1
```

After a file or device has been **OPENed** for **INPUT**, you can use the following instructions for reading the data stored in it:

INPUT#
Reads a string of data to a variable. Commas can be used to assign portions of the input to different variables. When reading from a sequential file, the records can be read one after the other by repeated **INPUT#** statements. The records are separated by commas in the string. Once a record has been read, it cannot be read again until the file has been **CLOSEd** and then **OPENed** again.

Example (reads six records in a file and places the data into six string variables):

```
10   OPEN "QFILE" FOR OUTPUT AS #1
20   PRINT #1, "Record A","a","b","c"
30   PRINT #1, "Record B",1,2,3
40   PRINT #1, "Record C","x";"y";"z"
50   PRINT #1, "Record D,Record E,Record F"
60   CLOSE #1
70   OPEN "QFILE" FOR INPUT AS #1
80   INPUT #1, A$
90   INPUT #1, B$
100  INPUT #1, C$
110  INPUT #1, D$,E$,F$
120  PRINT A$
130  PRINT B$
140  PRINT C$
150  PRINT D$
160  PRINT E$
170  PRINT F$
180  CLOSE #1
RUN
```

yields:

```
Record A   a          b          c
Record B   1          2          3
Record C   xyz
Record D
Record E
Record F
```

7.4 Input from a Sequential File, cont'd.

INPUT$
Reads a specified number of characters from the specified sequential file or channel. (If no file or channel is specified, the data on the standard IN channel will be read). The execution is held up waiting for the specified number of characters to be received. If a file does not contain as many characters as specified in the **INPUT$** statement, the execution will be resumed as soon as all available characters in the file have been received.

Sequential files are read from the start and once a number of characters have been read, they cannot be read again until the file is **CLOSEd** and **OPENed** again. Subsequent **INPUT$** statements will start with the first of the remaining available characters.

*Example (reads portions of characters from a file **OPENed** as #1):*

```
10    OPEN "QFILE" FOR OUTPUT AS #1
20    PRINT #1, "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30    CLOSE #1
40    OPEN "QFILE" FOR INPUT AS #1
50    A$=INPUT$(10,1)
60    B$=INPUT$(5,1)
70    C$=INPUT$(100,1)
80    PRINT "Record 1:",A$
90    PRINT "Record 2:",B$
100   PRINT "Record 3:",C$
110   CLOSE #1
RUN
```

yields:

```
Record1:  ABCDEFGHIJ
Record2:  KLMNO
Record3:  PQRTSUVWXYZ
```

LINE INPUT#
Works similar to **INPUT#**, but reads an entire line including all punctuation marks to a string variable instead of reading just one record. Note that commas inside a string will be regarded as punctuation marks and will not divide the string into records (compare with **INPUT#**).

Example (reads a complete line in a file and places the data into a single string variable):

```
10    OPEN "QFILE" FOR OUTPUT AS #1
20    PRINT #1, "Record A,Record B,Record C"
30    CLOSE #1
40    OPEN "QFILE" FOR INPUT AS #1
50    LINE INPUT #1, A$
60    PRINT A$
70    CLOSE #1
RUN
```

yields:

```
Record A,Record B,Record C
```

7.4 Input from a Sequential File, cont'd.

CLOSE

When a file is no longer used, it can be closed by means of a **CLOSE** statement containing the same reference number as the corresponding **OPEN** statement. An **END** statement also closes all open files.

A few instructions facilitate the use of files for sequential input:

☞ ***Relational Operators***
Also see:
• Chapter 4.9

EOF

The **EOF** function can connection with the statements **INPUT#**, **LINE INPUT#** and **INPUT$** to avoid the error condition *"Input past end"*. When the **EOF** function encounters the end of a file, it returns the value -1 (TRUE). If not, it returns the value 0 (FALSE).

Example:

```
10    DIM A%(10)
20    OPEN "DATA" FOR OUTPUT AS #1
30    FOR I%=1 TO 10
40    PRINT #1, I%*1123
50    NEXT I%
60    CLOSE #1
70    OPEN "DATA" FOR INPUT AS #2
80    I%=0
90    WHILE NOT EOF(2)
100   INPUT #2, A%(I%):PRINT A%(I%)
110   I%=1+1:WEND
120   IF EOF(2) THEN PRINT "End of File"
RUN
```

LOC (Location)

The **LOC** function returns the number of 128-byte blocks, that have been read or written since the file was **OPENed**.

This example closes the file "ADDRESSES" when record No. 100 has been read from the file:

```
10    OPEN "ADDRESSES" FOR INPUT AS #1
.....
.....
.....
200   IF LOC(1)=100 THEN CLOSE #1
.....
.....
```

LOF (Length-of-File)

The LOF function returns the length in bytes of an OPENed file.

The example illustrates how the length of the file "Pricelist" is returned:

```
10    OPEN "PRICELIST" AS #5
20    PRINT LOF(5)
.....
.....
```

7.5 Input from a Random File

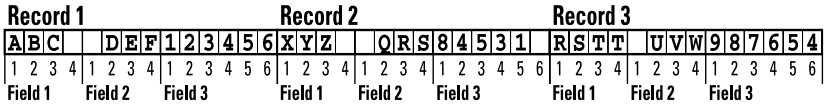
The following instructions are used in connection with input from random files:

OPEN	Creates and/or opens a file for **RANDOM** access and optionally specifies the record length in bytes.
FIELD	Creates a random buffer, divides it into fields and assigns a variable to each field.
GET	Reads a record from the buffer to the file.
CLOSE	Closes an **OPEN**ed file.
LOC	Returns the number of the last record read by the use of a **GET** statements in the specified file.
LOF	Returns the length in bytes of the specified file.

OPEN

To read the data stored in a random file, you must OPEN it.

The example in this chapter uses the random file created in chapter 8.4, which can be graphically illustrated like this:




```
10    OPEN "ZFILE" AS #1 LEN=14
```

The appending **LEN=14** refers to the length of each record which is 14 bytes (4 + 4 + 6). Do not confuse the **LEN** parameter in the **OPEN** statement with the **LEN** function, see chapter 9.2.

FIELD

Then enter the same field definitions as when the data was put into the file:

```
20    FIELD#1, 4 AS F1$, 4 AS F2$, 6 AS F3$
```

GET

Use a **GET** statement to copy the desired record from the file. Note that you can select whatever record you want, as opposed to sequential files, where you reads the records one after the other. In this case, we will copy record No. 1 (compare with the illustration above).

```
30    GET #1,1
```

If you like, you can copy data from other records in the same file by issuing additional **GET** statements with references to the records in question.

Now you can use the variables assigned to the fields in the record by means of the **FIELD** statement to handle the data. Possible numeric expressions converted to string format before being put into the record can now be converted back to numeric format using **VAL** functions. In our example, we will simply print the data on the screen:

```
40    PRINT F1$,F2$,F3$
```

☞ ***VAL function***
Also see:
• Chapter 9.2

7.5 Input from a Random File, cont'd.

CLOSE
Finally, close the file and execute:

```
50    CLOSE #1
RUN
```

yields:

```
ABC        DEF        123456
```

Two instructions facilitate the use of random files:

LOC (Location)
The **LOC** function returns the number of the last record read by the use of **GET** statement.

This example closes the file "ADDRESSES" when record No. 100 has been read from the file:

```
10    OPEN "ADDRESSES" AS #1
.....
.....
.....
200   IF LOC(1)=100 THEN CLOSE #1
.....
.....
```

LOF (Length-of-File)
The **LOF** function returns the length in bytes of an **OPENed** file.

The example illustrates how the length of the file "Pricelist" is returned:

```
10    OPEN "PRICELIST" AS #5
20    PRINT LOF(5)
. . . .
. . . .
```

7.6 Input from Printer's Keyboard

All UBI Fingerprint 7.xx-compatible EasyCoder printers are provided with a built-in keyboard containing a set of numeric keys supplemented with a number of function keys. There are also separate alphanumeric keyboards available as options[1].

Note that input from the printer's keyboard excludes the use of **ON KEY...GOSUB** statements (see chapter 5.8) and vice versa.

The following instructions are used in connection with input from the printer's keyboard:

OPEN	Opens the device "console:" for sequential **INPUT**.
INPUT#	Reads a string of data to a variable.
INPUT$	Reads a limited number of characters to a variable.
LINE INPUT#	Reads an entire line to a variable
CLOSE	Closes the device.

*[1]/. Input from an **external** alphanumeric keyboard is a case of ASCII input on a communication channel, see chapter 7.1-3.*

7.6 Input from Printer's Keyboard, cont'd.

The table below shows which ASCII characters the various keys will produce in unshifted and shifted position. However, the keyboard can be remapped (see later in this chapter).

Default ASCII decimal values

Key	Unshifted	Shifted	Notes
Shift	–	–	Adds 128 to the value of an unshifted key
F1	1	129	
F2	2	130	
F3	3	131	
F4	4	132	
F5	5	133	
C	8	136	
Enter	13	141	Unshifted Enter = Carriage Return
Feed	28	156	
Setup	29	157	
Pause	30	158	Shift+Pause is by default Break from keyboard
Print	31	159	
.	46	174	
0	48	176	
1	49	177	
2	50	178	
3	51	179	
4	52	180	
5	53	181	
6	54	182	
7	55	183	
8	56	184	
9	57	185	

The printable characters actually generated by the respective ASCII value depend on the selected character set (**NASC**/**NASCD**) and possible **MAP** statements, see chapter 9.1.

In case of **INPUT#** and **LINE INPUT#**, the input will not be accepted until a carriage return (<**Enter**>) is issued.

*This example demonstrates how the printable character and decimal ASCII value of various keys on the printer's keyboard can be printed to the screen of the host. You can break the program by holding down the <**Shift**> key and pressing <**Pause**>.*

```
10    PRINT "Character", "ASCII value"
20    OPEN "console:" FOR INPUT AS 1
30    A$=INPUT$(1,1)
40    B%=ASC(A$)
50    PRINT A$, B%
60    GOTO 30
70    CLOSE 1
RUN
```

7.7 Communication Control

The following instruction are used to control the communication between the printer and the host or other connected devices:

BUSY/READY	Transmits a busy or ready signal on the specified communication channel.
ON LINE/OFF LINE	Controls the **SELECT** signal on the parallel communication channel ("centronics:").
VERBON/VERBOFF	Turns printer's verbosity on/off.
SYSVAR(18)	Selects the printer's verbosity level.

☞ ***Communication***
Also see:
• Technical Manual, Setup Parameters

BUSY/READY

By means of these two statements, you can let the program execution turn a selected communication channel on or off. There is a difference between serial and parallel communication:

- ***Serial communication:***
  The type of busy/ready signal is decided in the Setup Mode (SerCom; Flowcontrol), see the Installation & Operation manual.
  - When a **BUSY** statement is executed, the printer sends a busy signal , e.g. XOFF or RTS/CTS low.
  - When a **READY** statement is executed, the printer sends a ready signal , e.g. XON or RTS/CTS high.
- ***Parallel communication:***
  The parallel Centronics communication channel uses the **BUSY**/**READY** statements to control the PE (paper end) signal on pin 12:
  - **BUSY** = PE high
  - **READY** = PE low

  The status of the PE signal can be read by a **PRSTAT** statement, e.g.:

```
IF (PRSTAT AND 4) GOTO.....ELSE GOTO.....
```

Note that issuing a **READY** statement is no guarantee that the printer will receive data, since there may be other conditions that hold up the reception, e.g. a full receive buffer.

ON LINE/OFF LINE

These two statements is only used for the parallel Centronics communication channel and controls the SELECT signal (pin 13 on the parallel interface board):

- **ON LINE 4** sets the SELECT signal high (default)
- **OFF LINE 4** sets the SELECT signal low

7.7 Communication Control, cont'd.

☞ ***Standard IN/OUT Channel***
Also see:
• Chapter 7.1

VERBON/VERBOFF
These two statements control the printer's verbosity, i.e. the response from the printer on the standard OUT channel to instructions received on the standard IN channel. Both can be substituted by **SYSVAR (18)**, see below.

By default, verbosity is on (**VERBON**). The verbosity level is controlled by the system variable **SYSVAR(18)**.

All responses will be turned suppressed when a **VERBOFF** statement is issued. However, **VERBOFF** does not suppress question marks and prompts displayed as a result of e.g. an **INPUT** statement. Instructions like **DEVICES**, **FILES**, **FONTS**, **IMAGES**, **LIST** and **PRINT** will also work normally.

SYSVAR
The system variable SYSVAR is used for many purposes, one of which is to control the verbosity level.

The verbosity level can be selected or read by specifying bits in **SYSVAR(18)**:

All levels enabled	*-1*
No verbosity	*0*
Echo received characters	*1*
"Ok" after correct command lines	*2*
*Echo **INPUT** characters from communication port*	*4*
Error after failed lines	*8*

The levels can be combined, so e.g. 3 means both *"Echo received characters"* and *"Ok after correct command line"*.

By default, all levels are enabled, i.e. **SYSVAR(18) = -1**

VERBON statement enables all levels, i.e. **SYSVAR(18) = -1**

VERBOFF statement disables all levels, i.e. **SYSVAR(18) = 0**

When the printer receives a character, e.g. from the keyboard of the host, by default the same character is echoed back on the standard OUT channel, i.e. usually to the screen of the host. When an instruction has been checked for syntax errors and accepted, the printer returns "Ok". Else an error message is returned.

This example demonstrates how the printer is set to only return "Ok" after correct lines (2) or error messages after failed lines (8):

```
SYSVAR(18) = 10
```

7.8 Background Communication

☞ ***Memory and Buffers***
Also see:
• Chapter 6.1

Background communication means that the printer receives data on an IN channel while the program runs in a loop. The data are stored in a buffer, that can be emptied at an appropriate moment by the running program, which then can use the data. Note that background communication buffers are not the same as the receive buffers. Any input received on a communication channel is first stored in the channel's receive buffer, awaiting being processed. After processing, the data may be stored in the background communication buffer.

The following instructions are used in connection with background communication:

COMSET	Decides how the background reception will work in regard of: - Communication channel. - Start character(s) of message string. - End character(s) of message string. - Characters to be ignored. - Attention string that interrupt reception. - Maximum number of characters to be received.
ON COMSET GOSUB	Branches the program execution to a subroutine when background reception on a specified channel is interrupted.
COMSET ON	Empties the buffer and turns on background reception on the specified channel.
COMSET OFF	Turns off background reception on the specified channel and empties the buffer.
COM ERROR ON	Enables error handling on a specified channel.
COM ERROR OFF	Disables error handling on a specified channel (default).
COMSTAT	Reads the status of the buffer of a specified channel.
COMBUF$	Reads data in the buffer of a specified channel.

To set up the printer for background communication, proceed as follows:

- Start by enabling the error handling for background communication using a **COM ERROR ON** statement and specifying the communication channel you intend to use:
  0 = "console:"
  1 = "uart1:"
  2 = "uart2:"
  4 = "centronics:"

7.8 Background Communication, cont'd.

- It may be useful to create a few messages indicating what have caused the interruption.

  Example:
  Error handling is enabled for communication channel "uart1:" and messages will be printed to the standard out channel for all conditions that can be detected by a ***COMSTAT*** *function.*

```
10    COM ERROR 1 ON
20    A$="Max. number of characters"
30    B$="End char. received"
40    C$="Communication error"
50    D$="Attention string received"
```

- Continue with a **COMSET** statement specifying:
  - Which communication channel will be used (0–4, see above).
  - Which character, or string of characters, will be used to tell the printer to start receiving data?
  - Which character, or string of characters, will be used to tell the printer to stop receiving data?
  - Which character or characters should be ignored, i.e. filtered out from the received data?
  - Which character, or string of characters, should be used as an attention string, i.e. to interrupt the reception.

  Start, stop, ignore and attention characters are selected according to the protocol of the computing device that transmits the data. Non printable characters, e.g. STX (Start of Text; ASCII 02 dec.) and ETX (End of Text; ASCII 03 dec.) can be selected by means of a ***CHR$*** *function. To specify no character, use an empty string, i.e. "".*

  - How many characters should be received before the transmission is interrupted? This parameter also decides the size of the buffer, i.e. how much of the temporary memory will be allocated.

  Example (designed to make the example easy to run rather than to illustrate a realistic application):
  Background reception on the serial channel "uart1:".
  Start character: A
  End character: CHR$ (90) i.e. the character "Z".
  Characters to be ignored: #
  Attention string: BREAK
  Max. number of characters in buffer: 20

```
60    COMSET 1,"A",CHR$(90),"#","BREAK",20
```

☞ ***CHR$ Function***
Also see:
• Chapter 9.2

7.8 Background Communication, cont'd.

- Decide what will happen, when the reception is interrupted, by specifying a subroutine to which the execution will branch, using an **ON COMSET GOSUB** statement. Interruption will occur when any of the following conditions is fulfilled:
  - an end character is received.
  - an attention string is received.
  - the maximum number of characters have been received.

  Example:
  When the reception of data on communication channel 1 ("uart1:") is interrupted, the execution will branch to a subroutine starting on line number 1000.

```
70    ON COMSET 1 GOSUB 1000
```

- After returning from the subroutine, use a **COMSET ON** statement to empty the buffer and turn on background reception again. e.g.:

```
80    COMSET 1 ON
```

- When the reception has been interrupted, it is time to see what the buffer contains. You can read the content of the buffer, e.g. to a string variable, using a **COMBUF$** function:

```
1000  QDATA$=COMBUF$(1)
```

- The **COMSTAT** function can be used to detect what has caused the interruption. Use the logical operator **AND** to detect the following four reason of interruption as specified by **COMSET**:
  - Max. number of characters received (2).
  - End character received (4).
  - Attention string received (8).
  - Communication error (32).

  Example:
  The various cases of interruption makes different messages to be printed to the standard OUT channel.

```
1010  IF COMSTAT(1) AND 2 THEN PRINT A$
1020  IF COMSTAT(1) AND 4 THEN PRINT B$
1030  IF COMSTAT(1) AND 8 THEN PRINT C$
1040  IF COMSTAT(1) AND 32 THEN PRINT D$
```

- If you want to temporarily turn off background reception during some part of the program execution, you can issue a **COMSET OFF** statement and then turn off the background reception again using a new **COMSET ON** statement.

  *Note that any **COMSET ON/OFF** statement empties the buffer and the content will be lost if you do not read it first, using a **COMBUF$** function.*

7.8 Background Communication, cont'd.

- After adding a few lines to print the content of the buffer (line 1050) and to create a loop that waits from input from the host (line 90), the entire example will look like this. You can run the example by typing **RUN** and pressing <**Enter**> on the keyboard of the host. Then enter various characters and see what happens, comparing with the start character, stop character, ignore character, attention string, and max. number of characters parameters in the **COMSET** statement.

```
NEW
10    COM ERROR 1 ON
20    A$="Max. number of char. received"
30    B$="End char. received"
40    C$="Attn. string received"
50    D$="Communication error"
60    COMSET 1, "A",CHR$(90),"#","BREAK",20
70    ON COMSET 1 GOSUB 1000
80    COMSET 1 ON
90    IF QDATA$="" THEN GOTO 90
100   END
1000  QDATA$=COMBUF$(1)
1010  IF COMSTAT(1) AND 2 THEN PRINT A$
1020  IF COMSTAT(1) AND 4 THEN PRINT B$
1030  IF COMSTAT(1) AND 8 THEN PRINT C$
1040  IF COMSTAT(1) AND 32 THEN PRINT D$
1050  PRINT QDATA$
1060  RETURN
RUN
```

Two instructions facilitate the use of background communication:

LOC (Locate)
The **LOC** function returns the status of the receive or transmitter buffers in an **OPENed** communication channel:

- If the channel is **OPENed** for **INPUT**, the remaining number of characters (bytes) to be read from the receive buffer is returned.
- If the channel is **OPENed** for **OUTPUT**, the remaining free space (bytes) in the transmitter buffer is returned.

The number of bytes includes characters that will be **MAPped** as NULL.

This example reads the number of bytes which remains to be received from the receiver buffer of "uart2:":

```
10    OPEN "uart2:" FOR INPUT AS #2
20    A%=LOC(2)
30    PRINT A%
...
...
```

7.8 Background Communication, cont'd.

LOF (Length-of-File)
The LOF function returns the status of the buffers in an OPENed communication channel:

- If a channel is **OPENed** for **INPUT**, the remaining free space (bytes) in the receive buffer is returned.
- If a channel is **OPENed** for **OUTPUT**, the remaining number of characters to be transmitted from the transmitter buffer is returned.

The example shows how the number of free bytes in the receive buffer of communication channel "uart2:" is calculated:

```
10    OPEN "uart2:" FOR INPUT AS #2
20    A%=LOF(2)
30    PRINT A%
...
...
80    COMSET 1 ON
90    IF QDATA$="" THEN GOTO 90
100   END
1000  QDATA$=COMBUF$(1)
1010  IF COMSTAT(1) AND 2 THEN PRINT A$
1020  IF COMSTAT(1) AND 4 THEN PRINT B$
1030  IF COMSTAT(1) AND 8 THEN PRINT C$
1040  IF COMSTAT(1) AND 32 THEN PRINT D$
1050  PRINT QDATA$
1060  RETURN
RUN
```

7.9 RS 422 Communication

As an option, the printers can be fitted with interfaces board that provides either RS 422 non-isolated or RS 422 isolated on "uart2:".

In neither of these protocols, there are any lines for hardware handshake (RTS/CTS).

RS 422 is a point-to-point four-line screened cable connection between a host computer and a printer, or between two printers. Two lines transmit data and the other two receive data. No hardware handshake can be used (4 lines only), but XON/XOFF or ENQ/ACK can be used if so desired.

- Fit straps and driver circuits according to the installation instructions for the interface board.
- Set the printer's flowcontrol setup to:

RTS/CTS:	Always Disable
ENQ/ACK:	Enable or Disable
XON/XOFF, Data to host:	Always Enable
XON/XOFF, Data to host:	Enable or Disable

- Select "uart2:" as standard I/O channel, e.g. **SETSTDIO 2,2**

7.10 External Equipment

Industrial Interface
The UBI Fingerprint firmware not only allows you to control the printer, but various types of external equipment, like conveyor belts, gates, turnstiles, control lamps etc. can be controlled as well by the program execution. Likewise, the status of various external devices can be used to control both the printer and other equipment. The computing capacity of the UBI Fingerprint printer can thus be used to independently control workstations without the requirement of an on-line connection to a host computer.

What makes this possible is the Industrial Interface Board, which is available as an option. The board contains a female DB-44 connector with 8 digital IN ports, 8 digital OUT ports and 4 OUT ports with relays.

There are two instruction solely used in connection with the Industrial Interface Board:

PORTOUT ON/OFF
This statement sets one of the four relays OUT ports or one of the digital OUT ports to either on or off.

PORTIN
This function returns the status of a specified port.

Refer to the installation instructions of the Industrial Interface Board for more details.

8. Output from UBI Fingerprint

8.1 Output to Std OUT Channel

The std. OUT channel is used for returning the printer's responses to instructions received from the host. That is why the same device usually is selected both standard IN and OUT channel (see **SETSTDIO** statement in chapter 7.1). By default, "uart1:" is std OUT channel.

☞ *Input to UBI Fingerprint*
See:
• Chapter 7

After every instruction received on the std IN channel, the printer will either return “Ok” or an error message (e.g. “Feature not implemented” or “Syntax Error”) on the std. OUT channel. If the std OUT channel is connected to the host computer, this message will appear on the screen.

☞ *Standard Error-Handling*
Also see:
• Chapter 16.1

The response can be turned off/on by means of **VERBOFF**/**VERBON** statements, the verbosity level can be selected by **SYSVAR(18)**, and the type of error message can be selected by **SYSVAR(19)**.

☞ *Verbosity*
Also see:
• Chapter 7.7

Some instructions return data on the std OUT channel only:

DEVICES	Lists all devices (also see chapter 4.10).
FILES	Lists all files in the current directory or another specified directory (also see chapter 6.2).
FONTS	Lists all fonts in the printer's entire memory (also see chapter 12.4).
IMAGES	Lists all images in the printer's entire memory (also see chapter 14.4).
LIST	Lists the current program in its entity or within a specified range of lines (also see chapter 5.4.
PRINT	Prints the content of numeric or string expressions and the result of functions and calculations (see below).
PRINTONE	Prints characters entered as ASCII values (see below).

PRINT (or ?)

The **PRINT** statement prints a line on the std OUT channel, i.e. usually the screen of the host. The **PRINT** statement can be followed by one or several expressions (string and/or numeric).

If the **PRINT** statement contains several expressions, these must be separated by either commas (,) semicolons (;), or plus signs (+, only between string expressions):

- A comma places the expression that follows at the start of next tabulating zone (each zone is 10 characters long).

  Example:

```
PRINT "Price","$10"                    yields:
Price     $10
```

8.1 Output to Std OUT Channel, cont'd.

PRINT (or ?), cont'd.

- A semicolon places the expression that follows immediately adjacent to the preceding expression.

  Example:

```
PRINT "Price_";"$10"                                    yields:
Price_$10
```

- A plus sign places the string expression that follows immediately adjacent to the preceding string expression (plus signs can only be used between two string expressions)'

  Example:

```
PRINT "Price_"+"$10"                                    yields:
Price_$1
```

- Each line is terminated by a carriage return, as to make the next **PRINT** statement being started on a new line. However, if a **PRINT** statement is appended by a semicolon, the carriage return will be suppressed and next **PRINT** statement will be printed adjacently to the preceding one.

  Example:

```
10 PRINT "Price_";"$10";
20 PRINT "_per_dozen"
RUN                                                     yields:
Price_$10_per_dozen
```

- A **PRINT** statement can also be used to return the result of a calculation or a function.

  Example:

```
PRINT 25+25:PRINT CHR$ (65) Yields:
50
A
```

- If the **PRINT** statement is not followed by any expression, a blank line will be produced.

PRINTONE

The **PRINTONE** statement prints the alphanumeric representation of one or several characters specified by their respective ASCII values (according to the currently selected character set, see **NASC** statement in chapter 9.1) to the standard OUT channel.

The **PRINTONE** statement is useful e.g. when a certain character cannot be produced from the keyboard of the host.

PRINTONE is very similar to the **PRINT** statement and follows the same rules regarding separating characters, i.e. commas and semicolons).

Example:

```
PRINTONE 80;114;105;99;101,36;32;49;48          yields:
Price     $_10
```

8.2 Redirecting Output from Std Out Channel to File

As described in chapter 8.1, by default some instructions return data on the standard OUT channel. However, it is possible to redirect such output to a file using the **REDIRECT OUT** statement, as described below.

REDIRECT OUT
This statement can be issued with or without an appending string expression:

- **REDIRECT OUT <sexp>**
  The string expression specifies the name of a sequential file that will be created and in which the output will be stored. Obviously, in this case no data will be echoed back to the host.
- **REDIRECT OUT**
  When no file name appends the statement, the output will be directed back to the std. OUT channel.

Example:
The output is redirected to the file "IMAGES.DAT". Then the images in the printer's memory is read to the file after which the output is redirected back to the standard OUT channel. Then the file is copied to the communication channel "uart1:" and printed on the screen of the host.

```
10    REDIRECT OUT "IMAGES.DAT"
20    IMAGES
30    REDIRECT OUT
RUN
Ok

COPY "IMAGES.DAT","uart1:"
```

yields e.g.:

```
CHESS2X2.1                CHESS4X4.1
DIAMONDS.1                UBI.1
UBI.2                     UBI010.1
UBI010.2

1543700 bytes free   307200 bytes used
Ok
```

8.3 Output and Append to Sequential Files

The following instructions are used in connection with output to sequential files:

OPEN	Creates and/or opens a file for sequential **OUTPUT** or **APPEND** and optionally specifies the record length in bytes.
PRINT#	Prints data entered as numeric or string expressions to the specified file.
PRINTONE#	Prints data entered as ASCII values to the specified file.
CLOSE	Closes an **OPENed** file.
LOC	Returns the number of 128-byte blocks, that have been written since the file was **OPENed**.
LOF	Returns the length in bytes of the specified file.

To print data to a sequential file, proceed as follows:

OPEN
Before any data can be written to a sequential file, it must be opened. Use the **OPEN** statement to specify the name of the file and the mode of access (**OUTPUT** or **APPEND**).

- **OUTPUT** means that existing data will be replaced.
- **APPEND** means that new data will be appended to existing data.

In the **OPEN** statement you must also assign a number to the **OPENed** file, which is used when the file is referred to in other instructions. The number mark (#) is optional. Optionally, the length of the record can also be changed (default 128 bytes). Up to 10 files and devices can be open at the same time.

Examples:
The file "ADDRESSES" is opened for output and given the reference number 1:

```
OPEN "ADDRESSES FOR OUTPUT AS #1
```

The file "PRICELIST" is opened for append and is given the reference number 5:

```
OPEN "PRICELIST" FOR APPEND AS #2
```

After a file or device has been **OPENed** for **OUTPUT** or **APPEND**, you can use the following instructions for writing data to it:

PRINT#
Prints data entered as string or numeric expressions to a sequential file. Expressions can be separated by commas or semicolons:

- Commas prints the expression in separate zones.
- Semicolons prints expressions adjacently.

There are two ways to divide the file into records:

- Each **PRINT#** statement creates a new record (see line 20-40 in the example below).
- Commas inside a string divides the string into records (see line 50 in the example below).

8.3 Output and Append to Sequential Files, cont'd.

PRINT#, cont'd.
Example:

```
10    OPEN "QFILE" FOR OUTPUT AS #1
20    PRINT #1, "Record A", "a", "b", "c"
30    PRINT #1, "Record B", 1, 2, 3
40    PRINT #1, "Record C", "x"; "y"; "z"
50    PRINT #1, "Record D,Record E,Record F"
```

PRINTONE#
Prints characters entered as decimal ASCII values according to the selected character set to the selected file or device. This statement is e.g. useful when the host cannot produce certain characters. Apart from using ASCII values instead of string or numeric expressions, the **PRINTONE#** works in the same way as the **PRINT#** statement.

Example (prints two records "Hello" and "Goodbye" to "file1"):

```
10    OPEN "file1" FOR OUTPUT AS 55
20    PRINTONE#55,72;101;108;108;111
30    PRINTONE#55,71;111;111;100;98;121;101
```

CLOSE
After having written all the data you need to the file, **CLOSE** it using the same reference number as when it was **OPENed**.

Example:

```
10    OPEN "file1" FOR OUTPUT AS 55
20    PRINTONE#55,72;101;108;108;111
30    PRINTONE#55,71;111;111;100;98;121;101
40    CLOSE 55
```

LOC (Location)
The **LOC** function returns the number of 128-byte blocks, that have been written since the file was **OPENed**.

This example closes the file "ADDRESSES" when record No. 100 has been read from the file:

```
10    OPEN "ADDRESSES" FOR OUTPUT AS #1
.....
.....
200   IF LOC(1)=100 THEN CLOSE #1
.....
```

LOF (Length-of-File)
The **LOF** function returns the length in bytes of an **OPENed** file.

The example illustrates how the length of the file "Pricelist" is returned:

```
10    OPEN "PRICELIST" FOR OUTPUT AS #5
20    PRINT LOF(5)
.....
.....
```

8.4 Output to Random Files

The following instructions are used in connection with output to random files:

OPEN	Creates and/or opens a file for **RANDOM** access and optionally specifies the record length in bytes.
FIELD	Creates a random buffer, divides it into fields and assigns a variable to each field.
LSET/RSET	Places data left- or right-justified into the buffer.
PUT	Writes a record from the buffer to the file.
CLOSE	Closes an **OPENed** file.
LOC	Returns the number of the last record written by the use of a **PUT** statement in the specified file.
LOF	Returns the length in bytes of the specified file.

To write data to a random file, proceed as follows:

OPEN
Start by **OPENing** a file for random input/output. Since random access is selected by default, the mode of access can be omitted from the statement, e.g.:

```
10    OPEN "ZFILE" AS #1
```

Optionally, the length of each record in the file can be specified in number of bytes (default 128 bytes):

```
10    OPEN "ZFILE" AS #1 LEN=14
```

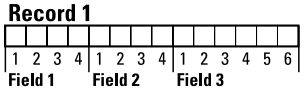
FIELD
Next action to take is to create a buffer by means of a **FIELD** statement. The buffer is given a reference number and divided into a number of fields with individual length in regard of number of characters. To each field, a string variable is assigned.

The buffer specifies the format of each record in the file. The sum of the length of the different fields in a record must not exceed the record length specified in the **OPEN** statement.

In the example below, 4 bytes are allocated to field 1, 4 bytes to field 2 and 6 bytes to field 3. The fields are assigned to the string variables A1$, A2$ and A3$ respectively.

```
20    FIELD#1, 4 AS F1$, 4 AS F2$, 6 AS F3$
```

Graphically illustrated, the record produced in the line above will look like this:



The file can consist of many records, all with the same format. (To produces files with different record lengths, the file must be **OPENed** more than once and with different reference numbers).

8.4 Output to Random Files, cont'd.

Now it is time to write some data to the file. Usually the data comes from e.g. the host or from the printer's keyboard. In this example, we will type the data directly on the host and assign the data to string variables:

```
30    QDATA1$="ABC"
40    QDATA2$="DEF"
50    QDATA3$="12345678"
```

☞ *STR$ Function*
Also see:
• Chapter 9.2

Note that only string variables can be used. Possible numeric expressions must therefore be converted to strings by means of **STR$** functions.

LSET/RSET

There are two instructions for placing data into a random file buffer:

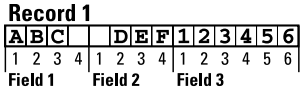
- **LSET** places the data left-justified.
- **RSET** places the data right-justified.

In other words, if the input data consist of less bytes that the field into which it is placed, it will either be placed to the left (**LSET**) or to the right (**RSET**).

If the length of the input data exceeds the size of the field, the data will be truncated from the end in case of **LSET**, and from the start in case of **RSET**.

```
60    LSET F1$=QDATA1$
70    RSET F2$=QDATA2$
80    LSET F3$=QDATA3$
```

Using the graphic illustration from previous page, the result is meant to be like this:



Note that the first field is left-justified, the second field is right-justified, and the third field is left-justified and truncated at the end (digits 7 and 8 are omitted since the field is only six bytes long; if the field had been right-justified, digits 1 and 2 had been omitted instead).

PUT

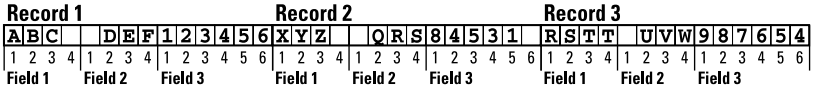
Next step is to transfer the record to the file. For this purpose we use the **PUT** statement. **PUT** is always followed by the number assigned to the file when it was **OPENed**, and the number of the record in which you want to place the data (1 or larger).

In our example, the file ZFILE was **OPENed** as #1 and we want to place the data in the first record. Note that you can place data in whatever record you like. The order is of no consequence.

```
90    PUT #1,1
```

8.4 Output to Random Files, cont'd.

If you want, you can continue and place data into other records using additional sets of **LSET**, **RSET** and **PUT** statements. Below is a graphic example of a three-record file:



CLOSE

When you are finished, close the file:

```
100 CLOSE #1
```

Nothing will actually happen before you execute the program using a **RUN** statement. Then the data will be placed into the fields and records as specified by the program, e.g.:

```
10    OPEN "ZFILE" AS #1 LEN=14
20    FIELD#1, 4 AS F1$, 4 AS F2$, 6 AS F3$
30    QDATA1$="ABC"
40    QDATA2$="DEF"
50    QDATA3$="12345678"
60    LSET F1$=QDATA1$
70    RSET F2$=QDATA2$
80    LSET F3$=QDATA3$
90    PUT #1,1
100   CLOSE #1
RUN
```

LOC (Locate)

The **LOC** function returns the number of the last record read or written by the use of **GET** or **PUT** statements respectively in an **OPENed** file.

This example closes the file "ADDRESSES" when record No. 100 has been read from the file:

```
10    OPEN "ADDRESSES" AS #1
.....
.....
200   IF LOC(1)=100 THEN CLOSE #1
.....
.....
```

LOF (Length-of-File)

The **LOF** function returns the length in bytes of an **OPENed** file.

The example illustrates how the length of the file "Pricelist" is returned:

```
10    OPEN "PRICELIST" AS #5
20    PRINT LOF(5)
.....
.....
```

8.5 Output to Communication Channels

Output from a UBI Fingerprint program can be directed to any serial communication channel **OPENed** for sequential **OUTPUT** following the same principles as for output to files (see chapter 8.3).

Note that in this case, the parallel communication channel "centronics:" cannot be used (one-way communication only).

The communication channels are specified by name as follows:
- "uart1:"
- "uart2:"

The following instructions are used in connection with output to a communication channel:

OPEN	Opens a serial communication channel for sequential output.
PRINT#	Prints data entered as numeric or string expressions to the selected channel.
PRINTONE#	Prints data entered as ASCII values to the selected channel.
CLOSE	Closes an **OPENed** channel.
LOC	Returns the remaining number of free bytes in the transmitter buffer of the selected communication channel.
LOF	Returns the remaining numbers of characters to be transmitted from the transmitter buffer is returned.
COPY	Copies a file to a communication channel.

Example 1 (prints the records “Record 1” and “Record 2” to the serial communication channel "uart2:"):

```
10    OPEN "uart2:" for OUTPUT AS #1
20    PRINT #1, "Record 1"
30    PRINTONE #1, 82;101;99;111;114;100;32;50
40    CLOSE #1
```

Example (prints the file "datafile" in a DOS-formatted memory card to the serial communication channel "uart2:"):

```
COPY "card1:datafile","uart2:"
```

8.6 Output to Display

The only device, other than the serial communication channels, that can be **OPENed** to receive output from a UBI Fingerprint program, is the printer's LCD display ("console:"). This is explained in chapter 15.2 together with other methods for controlling the display.

9. Data Handling

9.1 Preprocessing Input Data

All input data to the printer come in binary form via the various communication channels. Text files are transmitted in ASCII format, which upon reception will be preprocessed by the printer's software according to two instructions as to provide full compatibility between the printer and the host:

MAP	Remaps the selected character set.
NASC	Selects a single-byte character set
NASCD	Selects a double-byte character set

☞ *COMSET statement*
Also see:
• Chapter 7.8

☞ *ON KEY...GOSUB statement*
Also see:
• Chapter 15.1

A character received by the printer on a communication channel will first be processed in regard of possible **MAP** statements. Then the character will be checked for any **COMSET** or **ON KEY... GOSUB** conditions. When a character is to be printed, it will be processed into a bitmap pattern that makes up a certain character according to the character set selected by means of a **NASC** or **NASCD** statement.

MAP

The **MAP** statement is used to modify a character set or to filter out undesired characters on a specified communication channel by mapping them as Null (ASCII 0 dec).

If no character set meets your requirements completely (see **NASC** and **NASCD** below), select the set that comes closest and modify it using **MAP** statements. Do not map any characters to ASCII values occupied by characters used in UBI Fingerprint instructions, e.g. keywords, operators, %, $, #, and certain punctuation marks. Mapped characters will be reset to normal at power-up or reboot.

☞ *Character Sets*
Also see:
• UBI Fingerprint 7.xx Reference Manual for complete single-byte character set tables.

Example:
You may want to use the German character set (49) and 7 bits communication protocol. However, you need to print £ characters, but have no need for the & character. Then remap the £ character (ASCII 187 dec.) to the value of the & character (ASCII 38 dec.) . Type a series of & characters on the keyboard of the host and finish with a carriage return:

```
10 NASC 49
20 MAP 38,187
30 FONT "Swiss 721 BT"
40 PRPOS 100,100
50 INPUT "Enter character";A$
60 PRTXT A$
70 PRINTFEED
RUN
Enter character?
```

(see note!)

*Note! When using 7 bit communication, the printer cannot echo back the correct character to the host if its ASCII value exceeds 127, hence "**;**" characters will appear on the screen. Nevertheless, the desired "**£**" characters will be printed on the label.*

9.1 Preprocessing Input Data, cont'd.

NASC

The **NASC** statement is used to select a single-byte character set that decides how the various characters will be printed. This instruction makes it possible to adapt the printer to various national standards. By default, characters will be printed according to the Roman 8 character set.

Suppose you order the printer to print the character ASCII 124 dec. (We will not concern ourselves with how your computer and its keyboard are mapped. Refer to their respective manuals.) If you check the character set tables at the end of the UBI Fingerprint 7.xx Reference Manual, you will see that ASCII 124 will generate the character "|" according to the Roman 8 character set, "ù" according to the French character set and ñ according to the Spanish set etc. The same applies to a number of special national characters, whereas digits 0–9 and characters A–Z, a–z plus most punctuation marks are the same in all sets. Select the set that best matches your data equipment and printout requirements.

If none of the sets matches your requirements exactly, select the one that comes closest. Then, you can make final corrections by means of **MAP** statements, see above.

A **NASC** statement will have the following consequences:

- ***Text printing:***
  Text on labels etc. will be printed according to the selected character set. However, instructions concerning the printable label image, that already has been processed before the **NASC** statement is executed, will not be affected. This implies that labels may be multilingual.
- ***LCD display:***
  New messages in the display will be affected by a preceding **NASC** statement. However, a message that is already displayed will not be updated automatically. The display is able to show most printable latin characters.
- ***Communication:***
  Data transmitted from the printer via any of the communication channels will not be affected, as the data is defined by ASCII values, not as alphanumeric characters. The active character set of the **receiving** unit will decide the graphic presentation of the input data, e.g. on the screen of the host.
- ***Bar code printing:***
  The pattern of the bars reflects the ASCII values of the input data and is not affected by a **NASC** statement. The bar code interpretation (i.e. the human readable characters below the bar pattern) is affected by a **NASC** statement. However, the interpretation of bar codes, that have been processed and are stored in the print buffer, will not be affected.

This example selects the Italian character set:

```
NASC 39
```

9.1 Preprocessing Input Data, cont'd.

NASCD

The **NASCD** statement works similar to the **NASC** statement, but is used for double-byte character sets, i.e. for such fonts that require 2 bytes to specify a character according to the Unicode standard. This is for example the case with major Asian languages, such as Chinese, Korean and Japanese.

When a double-byte character set has been selected, the firmware will usually treat all characters from ASCII 161 dec. to ASCII 254 dec (ASCII A1 – FE hex) as the first part of a two-byte character. Next character byte received will specify the second part. However, the selected Unicode double-byte character set may specify some other ASCII value as the breaking point between single and double byte character sets.

There are various ways to produce double-byte characters from the keyboard of the computer. By selecting the proper character set using a **NASCD** statement, the typed-in ASCII values will be translated to the corresponding Unicode values, so the desired glyph will be printed.

Double-byte fonts and character set tables are available from UBI on special request, usually in the form of font cards.

Example:
The text field in line 50 contains both single- and double-byte fonts. The double-byte font and its character set are stored in a Font Install Card. The program yields a printed text line that starts with the Latin character A (ASCII 65 dec.) followed by the Chinese font that corresponds to the address 161+162 dec. in the character set "BIG5.NCD".

```
10    NASC 46
20    FONT "Swiss 721 BT", 24, 10
30    NASCD "rom:BIG5.NCD"
40    FONTD "Chinese"
50    PRTXT CHR$(65);CHR$(161);CHR$(162)
60    PRINTFEED
RUN
```

9.2 Input Data Conversion

There are a number of instruction for converting data in numeric or string expressions. You will find them used in many examples in this volume. The instructions will only be described in short terms. For full information, please refer to the UBI Fingerprint Reference Manual.

ABS

The **ABS** function returns the absolute value of a numeric expression. Absolute value means that the value is either positive or zero.

Example:

```
PRINT ABS (10-15)                    yields:
5
```

ASC

The **ASC** function returns the decimal ASCII value of the first character in a string expression.

Example:

```
PRINT ASC("HELLO")                   yields:
72
```

CHR$

The **CHR$** function returns the readable character from a decimal ASCII value. This function is useful when you cannot produce a certain character from the keyboard of the host.

Example:

```
PRINT CHR$(72)                       yields:
H
```

INSTR

The **INSTR** function searches a string expression for a certain character, or sequence of characters, and returns the position.

Example:

```
PRINT INSTR ("UBI","BI")             yields:
2
```

LEFT$

The **LEFT$** function returns a certain number of characters from the left side of a string expression, i.e. from the start. The complementary instruction is **RIGHT$**.

Example:

```
PRINT LEFT$("UBI PRINTER",3)         yields:
UBI
```

9.2 Input Data Conversion, cont'd.

LEN

The **LEN** function returns the number of characters including space characters in a string expression.

Example:

```
PRINT LEN ("UBI PRINTER")                    yields:
11
```

MID$

The **MID$** function returns a part of a string expression. You can specify start position and, optionally, the number of characters to be returned.

Example:

```
PRINT MID$ ("UBI PRINTER",5,2)               yields:
PR
```

RIGHT$

The **RIGHT$** function returns a certain number of characters from the right side of a string expression, i.e. from the end. The complementary instruction is **LEFT$**.

Example:

```
PRINT RIGHT$("UBI PRINTER",7)                yields:
PRINTER
```

SGN

The **SGN** function returns the sign (1 = positive, -1 = negative or 0 = zero) of a numeric expression.

Example:

```
PRINT SGN(5-10)                              yields:
-1
```

SPACE$

The **SPACE$** function returns a specified number of space characters and is e.g. useful for creating tables with monospace characters.

Example:

```
10    FONT "Swiss 721 BT"
20    X%=100 : Y%=300
30    FOR Q%=1 TO 5
40    INPUT "Commodity: ", A$
50    INPUT "Price $:", B$
60    C$=SPACE$(15-LEN(A$))
70    PRPOS X%,Y%
80    PRTXT A$+C$+"$ "+B$
90    Y%=Y%-40
100   NEXT
110   PRINTFEED
```

Note:
*When entering the price in the example for **SPACE$**, make sure to use a period character (.) to indicate the decimal point.*

9.2 Input Data Conversion, cont'd.

STR$

The **STR$** function returns the string representation of a numeric expression. The complementary instruction is **VAL**.

Example:

```
10    A%=123
20    A$=STR$(A%)
30    PRINT A%+A%
40    PRINT A$+A$
RUN                                                    yields:
246
123123
```

STRING$

The **STRING$** function returns a specified number of a single character specified either by its ASCII value or by being the first character in a string expression.

Example:

```
10    A$="*THE END*"
20    FIRST$=STRING$(4,42)
30    LAST$=STRING$(4,A$)
40    PRINT FIRST$+A$+LAST$
RUN                                                    yields:
*****THE END*****
```

VAL

The **VAL** function returns the numeric representation of a string expression. The complementary instruction is **STR$**.

VAL is for example used in connection with random files, which only accept strings (see chapters 7.5 and 8.4). Thus numeric expressions must be converted to string format using **STR$** before they are **PUT** in a random file and be converted back to numeric values using **VAL** after you **GET** them back from the file.

Another application is when you want to calculate using data in a string expression, e.g. when reading the printer's clock (also see chapter 9.3).

Example of how to use the printer as an alarm clock:

```
10    INPUT "Set Alarm"; A%
20    B%=VAL(TIME$)
30    IF B%>=A% THEN GOTO 40 ELSE GOTO 20
40    SOUND 880,100: END
RUN
```

9.3 Date and Time

☞ ***Setting Time and Date***
See:
• Chapter 15.5

The printer's CPU board is provided with a battery backed-up real-time clock (RTC).

The built-in calendar runs from 1980 through 2048 and corrects illegal values automatically, e.g. 971232 will be corrected to 980101.

The standard formats for date and time are:

Date **YYMMDD**, where...
YY are the two last digits of the year
MM are two digits representing the month (01–12)
DD are two digits representing the day (01–28|29|30|31)

Time **HHMMSS** where...
HH are two digits representing the hour (00-23)
MM are two digits representing the minute (00-59)
SS are two digits representing the second (00-59)

In addition to the standard formats, other formats for date and time can be specified by the following instructions:

FORMAT DATE$	Specifies the format of date strings returned by **DATE$** and **DATEADD$** instructions.
FORMAT TIME$	Specifies the format of date strings returned by **TIME$** and **TIMEADD$** instructions.
NAME DATE$	Specifies the names of the months.
NAME WEEKDAY$	Specifies the names of the weekdays.

The following instructions are used to read the clock/calendar:

<svar> = DATE$	Returns the current date in standard format to a string variable.
<svar> = DATE$("F")	Returns the current date in the format specified by **FORMAT DATE$** to a string variable.
<svar> = TIME$	Returns the current time in standard format to a string variable.
<svar> = TIME$("F")	Returns the current time in the format specified by **FORMAT TIME$** to a string variable.
DATEADD$	Adds or subtracts a number of days to/from the current date or a specified date and returns it in standard format, or the format specified by **FORMAT DATE$**.
TIMEADD$	Adds or subtracts a number of seconds to/from the current time or a specified moment of time and returns it in standard format, or the format specified by **FORMAT TIME$**.
DATEDIFF	Calculates the difference in days between two specified dates.
TIMEDIFF	Calculates the difference in seconds between two specified moments of time.
WEEKDAY	Returns the weekday of a specified date as a numeric constant (1–7).

9.3 Date and Time, cont'd.

WEEKDAY$	Returns the name of the weekday of a specified date in plain text according to the weekday names specified by **NAME WEEKDAY$**, or – if such a name is missing – the full name in English.
WEEKNUMBER	Returns the week number of a specified date.
TICKS	Returns the time passed since last startup in $^1/_{100}$ seconds.

Note that in most instructions, you can specify the current date or time by means of **DATE$** or **TIME$** respectively, e.g.:

```
WEEKDAY$ (DATE$)
TIMEDIFF (TIME$, "120000")
```

This example shows how the date and time formats are set and a table of the names of months is created. Finally, a number of date and time parameters are read and printed to the standard OUT channel after being provided with some explanatory text:

```
10    FORMAT DATE$ "MMM/DD/YYYY"
20    FORMAT TIME$ "hh.mm pp"
30    NAME DATE$ 1, "Jan":NAME DATE$ 2, "Feb"
40    NAME DATE$ 3, "Mar":NAME DATE$ 4, "Apr"
50    NAME DATE$ 5, "May":NAME DATE$ 6, "Jun"
60    NAME DATE$ 7, "Jul":NAME DATE$ 8, "Aug"
70    NAME DATE$ 9, "Sep":NAME DATE$ 10, "Oct"
80    NAME DATE$ 11, "Nov":NAME DATE$ 12, "Dec"
90    A%=WEEKDAY(DATE$)
100   PRINT WEEKDAY$(DATE$)+" "+DATE$("F")+" "
      +TIME$("F")
110   PRINT "Date:",DATE$("F")
120   PRINT "Time:",TIME$("F")
130   PRINT "Weekday:", WEEKDAY$(DATE$)
140   PRINT "Week No.:",WEEKNUMBER  (DATE$)
150   PRINT "Day No.:", DATEDIFF ("950101",DATE$)
160   PRINT "Run time:", TICKS\6000;" minutes"
170   IF A%<6 THEN PRINT "This is ";WEEKDAY$(DATE$);
      ". Go to work!"
180   IF A%>5 THEN PRINT "This is ";WEEKDAY$(DATE$);
      ". Stay home!"
RUN
```

yields e.g.:

```
Friday Jun/09/1995 08.00 am
Date:     Jun/09/1995
Time:     08.00 am
Weekday:  Friday
Week No.: 23
Day No.:  159
Run time: 1  minutes
This is Friday. Go to work!
```

9.3 Date and Time, cont'd.

*This example shows how the **TICKS** function is used to delay the execution for a specified period of time:*

```
10    INPUT "Enter delay in sec's: ", A%
20    B%=TICKS+(A%*100)
30    GOSUB 1000
40    END
1000  SOUND 440,50                                  (Start signal)
1010  IF B%<=TICKS THEN SOUND 880,100 ELSE GOTO 1010
1020  RETURN
RUN
```

9.4 Random Number Generation

The UBI Fingerprint software provides two instructions for generating random numbers, e.g. for use in test programs.

RANDOM

The **RANDOM** function generates a random integer within a specified interval.

This example tests a random dot on the printhead of a 12 dots/mm EasyCoder 501 XP printer:

```
10    MIN%=HEAD(-7)*85\100: MAX%=HEAD(-7)*115\100
20    DOTNO%=RANDOM(0,1279)
30    IF HEAD(DOTNO%)<MIN% OR HEAD(DOTNO%)>MAX% THEN
40    BEEP
50    PRINT "ERROR IN DOT "; DOTNO%
60    ELSE
70    BEEP
80    PRINT "HEADTEST: OK!"
90    END IF
RUN
```

RANDOMIZE

To obtain a higher degree of randomization, the random number generator can be reseeded using the **RANDOMIZE** statement. You can either include an integer with which the generator will be reseeded, or a prompt will appear asking you to do so.

This example prints a random pattern of dots after the random number generator has been reseeded:

```
10 RANDOMIZE
20 FOR Q%=1 TO 100
30 X%=RANDOM(50,400)
40 Y%=RANDOM(50,400)
50 PRPOS X%,Y%
60 PRLINE 5,5
70 NEXT
80 PRINTFEED
RUN
```

yields:

```
Random Number Seed (0 to 99999999) ?          (prompt)
```

10. Label Design

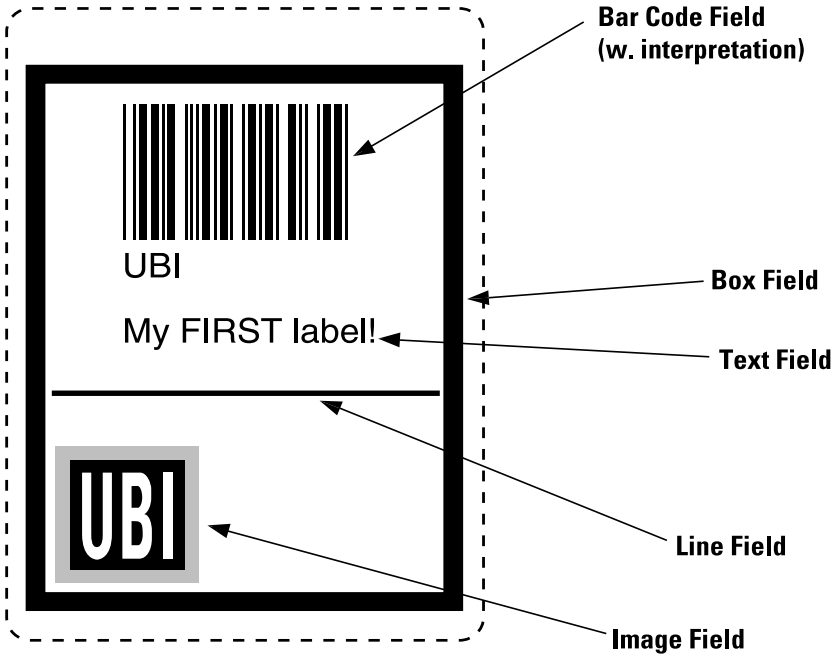
10.1 Creating a Layout

Field Types

A label layout is made up of a number of fields. There are five different types of fields:

- **Text Field** — A text field consists of a single line of text.
- **Bar Code Field** — A bar code field consists of a single bar code, with or without a bar code interpretation in human readable characters.
- **Image Field** — An image field is a picture, drawing, logotype or other type of illustration.
- **Box Field** — A box field is a square or rectangular paper-coloured area surrounded by a black border line. If the border is sufficiently thick, the whole area may appear black.
- **Line Field** — A line field is a black line that goes either along or across the paper web. A short but thick line can look like a black box.

There are no restrictions, other than the size of the printer's memory, regarding the number of fields on a single label.



10.1 Creating a Layout, cont'd.

☞ ***PRINTFEED Statement***
Also see:
• Chapter 11.3

☞ ***Setup Mode***
Also see:
• Chapter 15.6
• Installation & Operation manual

Origin

The positioning of all printable objects on the label, i.e. text fields, bar code fields, images, boxes, and lines, uses a common system. The starting point is called "origin" and is the point on the paper that corresponds to the innermost active dot on the printhead at the moment when the **PRINTFEED** statement is executed.

The location of the origo is affected by the following factors:

- ***Position across the paper web (X-axis):***
  The position of the origo is determined by the X-Start value in the Setup Mode .
- ***Position along the paper web (Y-axis):***
  The position of the origo is determined by the Feed adjustment in the Setup Mode and any **FORMFEED<nexp>** statements executed before the current **PRINTFEED** statement or after the preceding **PRINTFEED** statement.

Coordinates

Starting from the origin, there is a coordinate system where the X-axis runs across the paper web from left to right (as seen when facing the printer) and the Y-axis runs along the paper web from the printhead and towards the rear end of the paper.

Units of Measure

The unit of measure is always "dots", i.e. all measures depend on the density of the printhead. For example, in a printer with a 12 dots/mm printhead, a dot is $^1/_{12}$ mm = 0.0833 mm = 0.00328" or 3.28 mils. This implies that a certain label, originally designed for 12 dots/mm, will be printed larger in a 8 dots/mm printer. However, fonts are specified in points (not dots) and will thus be the same size regardless of printhead density.

A dot has the same size along both the X-axis and the Y-axis.

Insertion Point

The insertion point of any printable object is specified within this coordinate system by means of a **PRPOS<x-pos>,<y-pos>** statement. For example, the statement **PRPOS 100, 200** means that the object will be inserted at a position 100 dots to the right of the origin and 200 dots further back along the paper.

10.1 Creating a Layout, cont'd.

Alignment

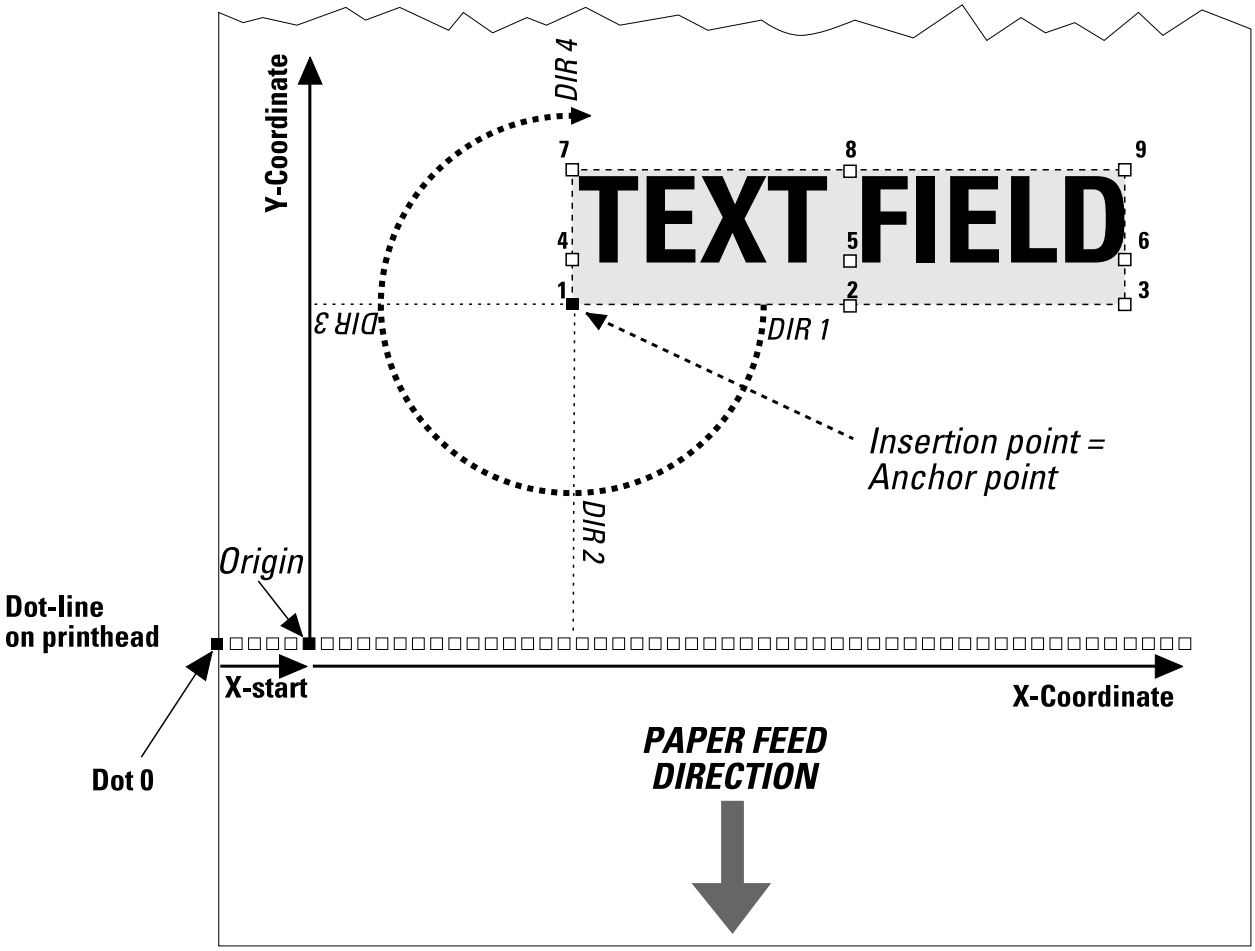
Once the insertion point is specified, you must also decided which part of the object should match the insertion point. For example, a text field forms a rectangle. There are 8 anchor points along the borders and one in the centre. The anchor points are numbered 1–9 and specified by means of an **`ALIGN`** statement. By specifying e.g. **`ALIGN 1`**, you will place the lower left corner of the text field at the insertion point specified by the **PRPOS** statement.

The illustration below shows the anchor points for the various types of fields. Refer to the UBI Fingerprint 7.xx Reference Manual, ALIGN statement for detailed information on the anchor points such bar codes, where the interpretation is an integrated part of the bar code pattern, e.g. EAN and UPC codes.

10.1 Creating a Layout, cont'd.

Directions

UBI Fingerprint allows printing in four different directions. Using a **`DIR`** statement, you can rotate the printable object clockwise around the anchor point/insertion point with a 90° increment (0°, 90°, 180°, or 270°), as illustrated below:



10.1 Creating a Layout, cont'd.

Layout Files

In addition to the method described above, there is an alternative method using files for specifying the various fields and their input data separately (see chapter 10.7). However, the various parameters of the layout file are based on the same principles as described in chapters 10.1 – 10.6.

Checking Current Position

After having positioned and specified an object, you can find out the current position of the insertion point by means of a **PRSTAT** function. This implies that after having e.g. entered a line of text, you can find out how long it will be and where any new object will be placed unless a new position is specified.

- In print direction 1 or 3, **PRSTAT (1)** returns the absolute value of the insertion point along the X-axis, whereas **PRSTAT (2)** returns the Y-value of the last executed **PRPOS** statement.
- In print direction 2 or 4, **PRSTAT (2)** returns the absolute value of the insertion point along the Y-axis, whereas **PRSTAT (1)** returns the X-value of the last executed **PRPOS** statement.

Example:
An unknown number of logotypes will be printed with 10 dots spacing across the paper web. The size of the logotype is not known. To avoid an "field out of label" error, a limitation in regard of paper width is included (line 80, change if necessary).

```
10    PRPOS 0,50
20    PRIMAGE "UBI.1"
30    X%=PRSTAT(1)
40    FOR A%=1 TO 10
50    Z%=PRSTAT(1)
60    PRPOS Z%+10,50
70    PRIMAGE "UBI.1"
80    IF Z%>550 THEN GOTO 100
90    NEXT
100   PRINTFEED
110   END
RUN
```

Note:
*The **PRSTAT** function can also be used for checking the printer's status in regard of a number of conditions, see chapter 16.3.*

10.2 Text Field

A text field consists of one or several alphanumeric characters on the same line (max 300 characters). UBI Fingerprint cannot wrap text to a new line, but each line must be specified as a separate text field.

In addition to the standard positioning statements **PRPOS**, **ALIGN** and **DIR**, a text field can contain the following instructions:

☞ ***Fonts***
Also see:
• Chapter 12

FONT (FT) and FONTD
Specifies the single- or double-byte font to be printed respectively. Default choice is the single-byte font Swiss 721 BT in 12 points size and with no slant. Once a font has been specified, it will be used in all text fields until a new **FONT** or **FONTD** statement is executed.

MAG
Fonts can be magnified 1 – 4 times independently in regard of height and width. This facility is mainly retained for compatibility with earlier UBI Fingerprint versions. The printout qulaity will be better if you specify a larger font size rather than magnifying a smaller one.

NORIMAGE (NI) / INVIMAGE (II)
Normally, text is printed in black on a paper-coloured background (**NORIMAGE**). Using **INVIMAGE**, the printing can be inversed so the paper gives the colour of the characters, whereas the background will be black. The size of the background is decided by the character cell. A **NORIMAGE** statement is only needed when changing back from **INVIMAGE** printing.

PRTXT (PT)
Text can be entered in the form of numeric expressions and/or string expressions. Two or more expression can be combined using semicolons (;) or, in case of string expressions, by plus signs (+). String constants must be enclosed by double quotation marks ("..."). Variables are useful for printing e.g. time, date or various counters, and when the same information is to appear in several places, e.g. both as plain text and as bar code input data.

NORIMAGE
INVIMAGE

10.2 Text Field, cont'd.

Summary

To print a text field, the following information and instructions must be given (default values will substitute missing parameters):

Purpose	Instruction	Default	Remarks
X/Y Position	PRPOS (PP)	0/0	Number of dots
Alignment	ALIGN (AN)	1	Select ALIGN 1 – 9
Direction	DIR	1	Select DIR 1 – 4
Typeface	FONT (FT)	Swiss 721 BT,12,0	
	FONTD	n.a.	
Magnification	MAG	1,1	Height 1 – 4, Width 1 – 4
Style	INVIMAGE (II)	no	White on black print
	NORIMAGE (NI)	yes	Black on white print
Text	PRTXT (PT)	n.a.	Field input data
Print a label	PRINTFEED (PF)	n.a.	Resets parameters to default

Example:

```
10    PRPOS 100,200
20    ALIGN 7
30    DIR 2
40    FONT "Swiss 721 Bold BT,10,15"
50    MAG 2,2
60    INVIMAGE
70    PRTXT "HELLO"
80    PRINTFEED
RUN
```

10.3 Bar Code Field

As standard, UBI Fingerprint 7.11 supports 40 of the most common bar code symbologies including two-dimensional bar codes and dot codes like PDF417, USD5, MaxiCode, and LEB. Each bar code (optionally including its human readable interpretation) makes up a bar code field.

In addition to the standard positioning statements **PRPOS**, **ALIGN** and **DIR**, a bar code field can contain the following instructions:

☞ ***Bar Codes***
Also see:
• Chapter 13

BARSET
This statement species the type of bar code and how it will be printed and can, if so desired, replace the following statements:

BARHEIGHT (BH)	Height of the bars in the code
BARRATIO (BR)	Ratio between wide and narrow bars
BARTYPE (BT)	Bar code type
BARMAG (BM)	Enlargement

The **BARSET** statement contains optional parameters for specifying complex 2-dimensional bar or dot codes, e.g. PDF417 (see UBI Fingerprint 7.xx Reference Manual).

For common one-dimensional bar codes the following parameters should be included in the statement:

• ***Bar code type***	Name must be given according to list in chapter 13.1 and be enclosed by double quotation marks ("..."). Default: "INT2OF5"
• ***Ratio (wide bars)***	Default: 3
• ***Ratio (narrow bars)***	Default: 1
• ***Enlargement***	Affects the bar pattern but not the interpretation, unless the bar font is an integrated part of the code, e.g. EAN/UPC. Default: 2
• ***Height***	Height of the bars in dots. Default: 100.

BARFONT...ON
Specifies the single-byte font to be used for the bar code interpretation (human readables). Note that in some bar codes (e.g. EAN/UPC) the interpretation is an integrated part of the code.

☞ ***Fonts***
Also see:
• Chapter 12

The bar font can be specified in regard of:

• ***Font***	Default: Swiss 721 BT
• ***Size in points***	Default: 12 points.
• ***Slant in degrees***	Default: 0.
• ***Vertical offset***	Specifies the distance in dots between the bottom of the bar pattern and the top of the interpretation characters. Default: 6.
• ***Height Magnification***	Default: 1
• ***Width Magnification***	Default: 1
• ***ON***	Enables the printing of the interpretation. Default: Disabled

3. Bar Code Field, cont'd.

BARFONT OFF

To disable bar code interpretation printing, use **BARFONT OFF**.

PRBAR (PB)

Input data to be used to generate the bar code can be entered in the form of a numeric or expressions. String constants must be enclosed by double quotation marks ("..."). Variables are useful for printing e.g. time, date or various counters, and when the same information is to appear in several places, e.g. both as plain text and as bar code input data.

Summary

To print a bar code field, the following information and instructions be must given (in most cases default values will substitute missing information):

Purpose	Instruction	Default	Remarks
X/Y Position	PRPOS (PP)	0/0	Number of dots
Alignment	ALIGN (AN)	1	Select ALIGN 1 – 9
Direction	DIR	1	Select DIR 1 – 4
Bar Code Select	BARSET	see above	
Hum. Readables	BARFONT...ON	see above	Can be omitted
Input Data	PRBAR (PB)	n.a.	Input data to bar code field
Print a label	PRINTFEED (PF)	n.a.	Resets parameters to default

Example:

```
10    PRPOS 50,500
20    ALIGN 7
30    DIR 4
40    BARSET "CODE39",2,1,3,120
50    BARFONT "Swiss 721 Bold BT,10,0",5,1,1 ON
60    PRBAR "UBI"
70    PRINTFEED
RUN
```

10.4 Image Field

☞ ***Image Downloading***
Also see:
• Chapter 14

An image field is a field containing a picture or logotype in .PCX format, which has been downloaded and installed in the printer.

In addition to the standard positioning statements **PRPOS**, **ALIGN** and **DIR**, an image field can contain the following instructions:

MAG
Images can be magnified 1-4 times independently in regard of height and width.

NORIMAGE (NI) / INVIMAGE (II)
Normally, images are printed as created, i.e. in black without any background (**NORIMAGE**). Using **INVIMAGE** the black and non-printed background can exchange colours. The size of the background is decided by the size of the image. A **NORIMAGE** statement is only needed when changing back from **INVIMAGE** printing.

PRIMAGE (PM)
Specifies the image by name in the form of a string expression. A string constant must be enclosed by double quotation marks ("..."). A string variable may be useful when the same image is to appear in several places. The extension indicates the suitable directions:
Extension .1 matches **DIR 1** and **DIR 3**
Extension .2 matches **DIR 2** and **DIR 4**

Summary
To print an image field, the following instructions must be given (in most cases default values will substitute missing information):

Purpose	Instruction	Default	Remarks
X/Y Position	PRPOS (PP)	0/0	Number of dots
Alignment	ALIGN (AN)	1	Select ALIGN 1 – 9
Direction	DIR	1	Select DIR 1 – 4
Magnification	MAG	1,1	Height 1 – 4, Width 1 – 4
Style	INVIMAGE (II)	no	White-on-black
	NORIMAGE (NI)	yes	Black-on-white
Image	PRIMAGE (PM)	n.a.	.1 or .2 depending on dir.
Print a label	PRINTFEED (PF)	n.a.	Resets parameters to default

Example:

```
10 PRPOS 50,50
20 ALIGN 9
30 DIR 3
40 MAG 2,2
50 INVIMAGE
60 PRIMAGE "UBI.1"
70 PRINTFEED
RUN
```

10.5 Box Field

A box is a hollow square or rectangle that can be rotated with an increment of 90° according to the print direction. If the line thickness is sufficiently large, the box will appear to be filled (another method is to print an extremely thick short line).

In addition to the standard positioning statements **PRPOS**, **ALIGN** and **DIR**, a box field can only contain the following instruction:

PRBOX (PX)
Specifies the size of the box in regard of height, width and line weight (thickness) in dots.

Summary
To print a box, the following information and instructions must be given (in some cases default values will substitute missing information):

Purpose	Instruction	Default	Remarks
X/Y Position	PRPOS (PP)	0/0	Number of dots
Alignment	ALIGN (AN)	1	Select ALIGN 1 – 9
Direction	DIR	1	Select DIR 1 – 4
Box spec:s	PRBOX (PX)	n.a.	Height, width and line weight in dots
Print a label	PRINTFEED (PF)	n.a.	Resets parameters to default

Example:

```
10 PRPOS 250,250
20 ALIGN 1
30 DIR 3
40 PRBOX 200,200,10
50 PRINTFEED
RUN
```

10.6 Line Field

A line can be printed in right angles along or across the paper according to the print direction.

In addition to the standard positioning statements **PRPOS**, **ALIGN** and **DIR**, a line field can only contain the following instruction:

PRLINE (PL)
Specifies the size of the line in regard of length and line weight (thickness) in dots.

Summary
To print a line, the following information and instructions must be given (in some cases default values will substitute missing information):

Purpose	Instruction	Default	Remarks
X/Y Position	PRPOS (PP)	0/0	Number of dots
Alignment	ALIGN (AN)	1	Select ALIGN 1 – 9
Direction	DIR	1	Select DIR 1 – 4
Line spec:s	PRLINE (PL)	n.a.	Length and width in dots
Print a label	PRINTFEED (PF)	n.a.	Resets parameters to default

Example:

```
10 PRPOS 100,100
20 ALIGN 1
30 DIR 4
40 PRLINE 200,10
50 PRINTFEED
RUN
```

10.7 Layout Files

Introduction

Many application, e.g. in connection with booking and ticketing, require the label layout as well as variable input data and logotypes to be sent to the printer as files or arrays. This method requires less programming in the printer and less data to be transferred between printer and host, but some kind of overhead program in the host, that handles file transfers as well as the input of data, is of great help.

The program instruction is a statement called **LAYOUT**. Before using this statement, a number of files or arrays must be created.

Creating a Layout File

The basis of the method is a layout file in random format, that contains a number of records, each with a length of 52 bytes. Each record can define:

- a line of fixed and/or variable text,
- a bar code with fixed and/or variable input data,
- bar code interpretation enable/disable and bar code font select,
- a logotype,
- a box, or
- a line.

Each record starts with a 2-byte hexadecimal element number (bytes 0–1) which is used to link the layout record with a variable input record or a record in a layout name file as explained later.

Byte 2 contains a single character that specifies the type of record:
A = Logotype (specified by its name)
B = Bar Code
C = Character (i.e. plain text)
H = Bar Code Font
L = Logotype (specified by its number)
S = Separation line
X = Box

The remaining bytes are used differently depending on type of record and specify e.g. direction, position, font etc. Each such instruction corresponds to a UBI Fingerprint instruction, e.g. direction corresponds to **DIR** statement, alignment to **ALIGN**, x- and y-positions to **PRPOS** etc. Note that there are only10 bytes available for the font and bar font names. Since most names of standard fonts are longer, you may need to use font aliases.

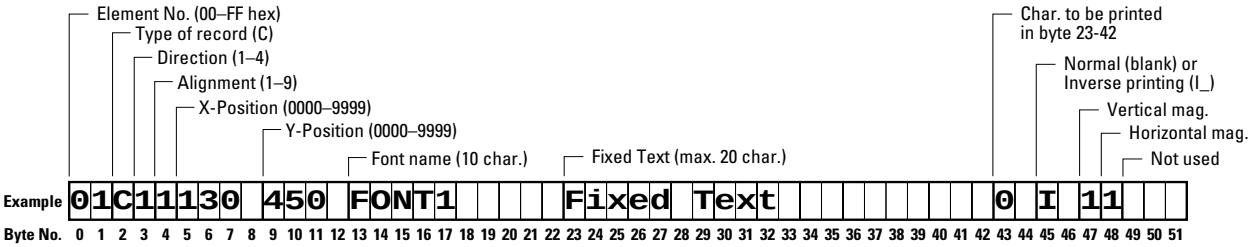
☞ ***Font Aliases***
See:
• Chapter 12.??

Text and bar code records can contain both fixed and variable data. The fixed data (max. 20 characters) are entered in the layout record. A parameter (bytes 43–44) specifies how many characters (starting from the first character) of the fixed data that will be printed or used to generate the bar code. Possible variable data will be appended to the fixed data at the position specified in bytes 43–44.

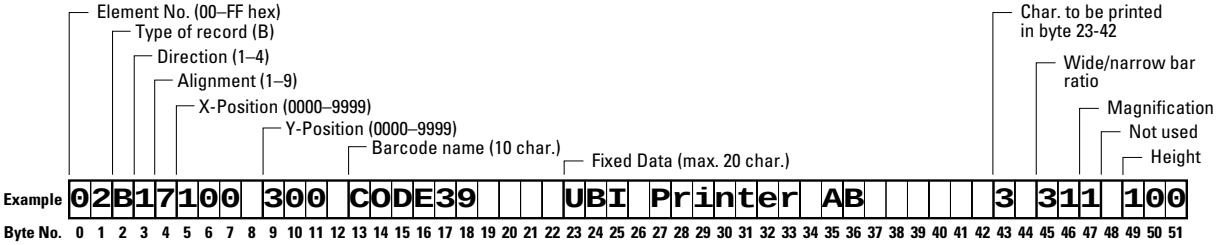
10.7 Layout Files, cont'd.

Creating a Layout File, cont'd.:
Syntax of layout file records for text and bar code printing:

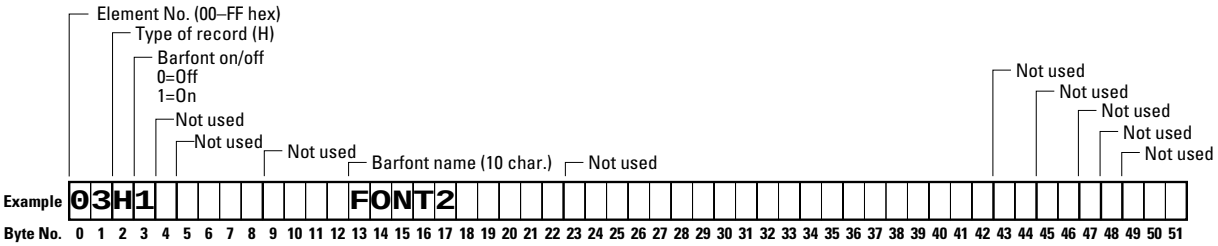
TEXT RECORD:



BAR CODE RECORD:



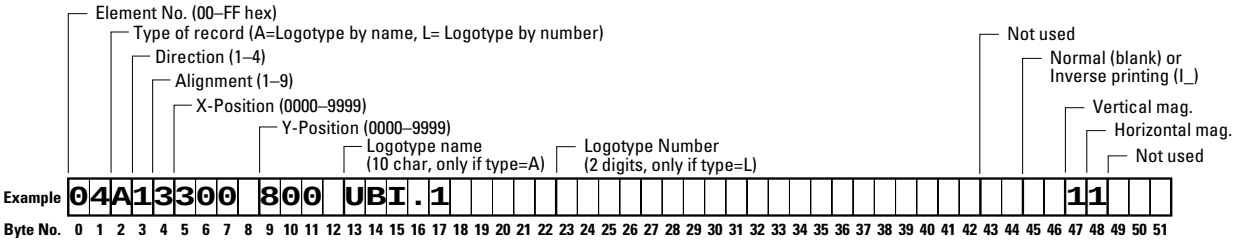
BAR CODE INTERPRETATION RECORD:



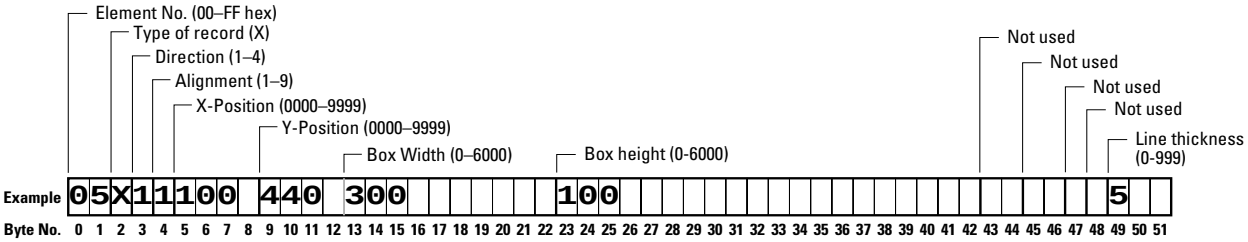
10.7 Layout Files, cont'd.

Creating a Layout File, cont'd.:
Syntax of layout file records for logotype, box and line printing:

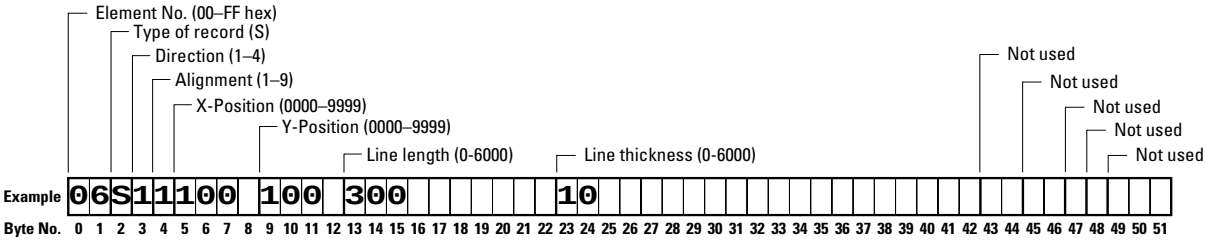
LOGOTYPE RECORD:



BOX RECORD:



LINE RECORD:



10.7 Layout Files, cont'd.

Creating a Layout File, cont'd.:
This example shows how a small layout file can be composed:

```
10 OPEN "LAYOUT.DAT" FOR OUTPUT AS 2                                    Open random file
20 PRINT #2, "01H1         FONT1                                  ";    Barfont record
30 PRINT #2, "02C11100 650 FONT1     Fixed Text          11I 22   ";    Text record
40 PRINT #2, "02C11130 450 FONT1     Fixed Text          0   11   ";    Text record
50 PRINT #2, "03B17100 300 CODE39    UBI                 3 311 100";    Bar code record
60 PRINT #2, "04A12300 800 UBI.1                            11   ";    Logotype record
70 PRINT #2, "05X11100 440 300       100                      5  ";    Box record
80 PRINT #2, "06S11100 100 300       10                         ";    Line record
90 CLOSE 2                                                              Close file
```

There are certain rules that should be observed:

- Each record must be exactly 52 bytes long and be appended by a semicolon (;).
- It is essential that the different types of data are entered exactly in the correct positions. Any input in unused bytes will be ignored.
- The records are executed in the order they are entered. The reference number at the start of each record does not affect the order of execution. This implies that a barfont record will affect all following bar code records, but not those already entered.
- When using bar code interpretation, do not enter a bar code record directly after a record with inverse printing, since the bar code interpretation will be inversed as well. A text or logotype record without inverse printing between the bar code record and the inversed record will reset printing to normal.

Creating a Logotype Name File
Next step is to create a logotype name file. This is a necessary step even if you are not going to use any logotype in your layout (in this case the file can be empty). In the layout file, you can set a logotype record to use logotypes specified either by name or by number.

- If you specify logotype-by-name (record type A), the printer's entire memory will be searched for an image with the specified name. A logotype-by-name file is composed by a number of records with a length of 10 bytes each that contain the image names, e.g.:

```
10   OPEN "LOGNAME.DAT" FOR OUTPUT AS 1
20   PRINT#1, "UBI.1     "
30   PRINT#1, "UBI.2     "
40   PRINT#1, "DIAMONDS.1"
50   PRINT#1, "DIAMONDS.2";
60   CLOSE 1
```

Note that the last record in a sequential file must be appended by a semicolon (;).

10.7 Layout Files, cont'd.

Creating a Logotype Name File, cont'd.:

- If you specify logotype-by-number (record type L), you must have a logotype name file. A logotype-by-number file is composed by a number of records with a length of 13 bytes each. The first 2 bytes is a reference number (0–99), the third byte is always a colon (:) and the following 10 bytes are used for the image name:

```
10    OPEN "LOGNAME.DAT" FOR OUTPUT AS 1
20    PRINT#1, "0 :UBI.1     "
30    PRINT#1, "1 :UBI.2     "
40    PRINT#1, "2 :DIAMONDS.1"
50    PRINT#1, "3 :DIAMONDS.2";
60    CLOSE 1
```

Note that the last record in a sequential file must be appended by a semicolon (;).

Creating a Data File or Array

You will also need a data file or data array. This file or array contains variable data that will be placed in the position specified by the layout. Each data record starts with a hexadecimal element number (00-FF hex) that links the data to the layout record or records that start with the same element number. Thus you can e.g. use a single data record to generate a number of text fields with various locations and appearances as well as to generate a bar code.

IMPORTANT!
*The **LAYOUT** statement requires that you use the same format (either files or arrays) for both data and errors.*

If you for some reason do not use variable data, you will still need to create either an empty data file or an empty data array.

- Create a data array like this:

```
10    DIM LAYDATA$(7)
20    LAYDATA$(0)="01Mincemeat"
30    LAYDATA$(1)="0AVeal"
40    LAYDATA$(2)="17Roast Beef"
50    LAYDATA$(3)="3FSausages"
60    LAYDATA$(4)="02Venison"
70    LAYDATA$(5)="06Lamb Chops"
80    LAYDATA$(6)="7CPork Chops"
```

☞ ***Arrays***
Also see:
• Chapter 6.10

- You can create a data file with the same content in a similar way:

```
10    OPEN "LAYDATA.DAT" FOR OUTPUT AS 1
20    PRINT#1,"01Mincemeat"
30    PRINT#1,"0AVeal"
40    PRINT#1,"17Roast Beef"
50    PRINT#1,"3FSausages"
60    PRINT#1,"02Venison"
70    PRINT#1,"06Lamb Chops"
80    PRINT#1,"7CPork Chops";
90    CLOSE 1
```

Note that the last record in a sequential file must be appended by a semicolon (;).

10.7 Layout Files, cont'd.

Creating an Error File or Array

The last requirement is an error file or array that can store any errors that may occur. If you use a data array, you must use an error array, and if you use a data file, you must use an error file. The following errors will be stored and presented in said order:

1 If an error occurs in a layout record, the number of the record (1...nn) and the error number is placed in the error array or file.

2 If a data record cannot be used in a layout record, an the index of the unused data record (0...nn) plus the error code -1 is placed in the error array or file.

- Error arrays must be large enough to accommodate all possible errors. Thus, use a **DIM** statement to specify a one-dimensional array with a number of elements that is twice the sum of all layout records plus twice the sum of all data records. You should also include some routine that reads the array, e.g.:

```
10    DIM QERR%(28)
20    QERR%(0)=0
.....
190   IF QERR%(1)=0 THEN GOTO 260
200   PRINT "-ERROR- LAYOUT 1"
210   I%=0
220   IF QERR%(I%)=0 THEN GOTO 260
230   PRINT "ERROR ";QERR%(I%+1);" in record ";QERR%(I%)
240   I%=I%+2
250   GOTO 220
260   PRINTFEED
```

- Error files require a little more programming to handle the error message, e.g.:

```
220 OPEN "ERRORS.DAT" FOR INPUT AS 10
230 IF EOF(10) THEN GOTO 280 ELSE GOTO 240
240 FOR A%=1 TO 28
250 INPUT #10, A$
260 PRINT A$
270 NEXT A%
280 PRINTFEED
```

Note that the loop in line 240 must be large enough to accommodate all possible errors.

☞ ***Arrays***
Also see:
• Chapter 6.10

10.7 Layout Files, cont'd.

Using the Files in a LAYOUT Statement

Now, you have all the files you need to issue a **LAYOUT** statement. This statement combines the layout file, the logotype file, the data file/array, and the error file/array into a printable image. Depending on whether you have selected to use data and error files or arrays, the statement will have a somewhat different syntax:

Files:

LAYOUT F, <layout file>, <logotype file>,<data file>,<error file>

Arrays:

LAYOUT <layout file>,<logotype file>,<data array>,<error array>

Note that you cannot omit any file or array, since the syntax requires a file name or array designation in each position. If you, for example, do not require any logotype, you must still create an empty logotype file.

Example:
The example below shows a simple layout created using the layout statement in combination with data and error arrays:

```
10 DIM QERR%(28)
20 LAYDATA$(0)="02Var. input"
30 LAYDATA$(1)="03 PRINTER"
40 QERR%(0)=0
50 OPEN "LOGNAME.DAT" FOR OUTPUT AS 1
60 PRINT #1, "UBI.1";
70 CLOSE 1
80 REM:LAYOUT FILE
90 OPEN "LAYOUT.DAT" FOR OUTPUT AS 2
100 PRINT #2, "01H1         FONT1                                    ";
110 PRINT #2, "02C11100 650 FONT1     Fixed Text          11I 22   ";
120 PRINT #2, "02C11130 450 FONT1     Fixed Text          0   11   ";
130 PRINT #2, "03B17100 300 CODE39    UBI                 3 311 100";
140 PRINT #2, "04A12300 800 UBI.1                             11   ";
150 PRINT #2, "05X11100 440 300       100                       5  ";
160 PRINT #2, "06S11100 100 300       10                           ";
170 CLOSE 2
180 LAYOUT "LAYOUT.DAT","LOGNAME.DAT",LAYDATA$,QERR%
190 IF QERR%(1)=0 THEN GOTO 260
200 PRINT "-ERROR- LAYOUT 1"
210 I%=0
220 IF QERR%(I%)=0 THEN GOTO 260
230 PRINT "   ERROR   "; QERR%(I%+1); " in record "; QERR%(I%)
240 I%=I%+2
250 GOTO 220
260 PRINTFEED
RUN
```

11. Printing Control

11.1 Paper Feed

In order to provide maximum flexibility, there are a number of instructions for controlling the paper feed without actually printing any labels:

CLEANFEED	Runs the printer's paper feed mechanism in order to facilitate cleaning of the print roller.
FORMFEED	Feeds out a blank label (or similar) or optionally feeds out or pulls back a specified amount of paper without printing.
TESTFEED	Adjusts the label stop sensor or black mark sensor while feeding out a number of blank labels (similar).
LBLCOND	Overrides the paper feed setup.

The paper is feed past the printhead by a rubber-coated roller driven by a stepper motor controoled by the firmware. The movement of the paper is detected by the label stop sensor (LSS) or black mark sensor (BMS), except when various types of paper strip are used.

The printer's setup in regard of *Media; Media Size; Length* and *Media; Media Type* is essential for how the paper feed will work. There are four or five different types of *Media Type* options (also see the Installation & Operation manual):

- Label (w gaps)
- Ticket (w mark)
- Ticket (w gaps)
- Fix length strip
- Var length strip

When a **FORMFEED**, **TESTFEED** or **PRINTFEED** statement is executed and the paper web is fed out, the photo-electric label stop sensor detects the front edge of each new label or the rear edge of each detection gap. Alternatively the black mark sensor detects the front edge of each black mark.

By performing a **TESTFEED** operation after loading a new supply of paper, the firmware is able to measure the distance between the forward edges of two consecutive labels, thereby determining the label length, and can adjust the paper feed accordingly. The same principle applies to tickets or tags with detection gaps and tickets with black marks.

☞ ***TESTFEED***
*To execute a **TESTFEED** at paper load, simultaneously press <**Shift**> + <**Feed**> on the printer's keyboard.*

11.1 Paper Feed, cont'd.

In case of paper strip, the LSS will only detect possible out-of-paper conditions, and the amount of paper feed is decided in two different ways:

- ***Fixed length strip***
  The amount of paper feed for each **FORMFEED**, **TESTFEED** and **PRINTFEED** operation is decided by the *Media; Media Size; Length* setup.
- ***Variable length strip***
  At the execution of a **PRINTFEED**, the firmware will add a sufficient amount of paper feed after the last printable object to allow the paper to be torn off. Note that e.g. a blank space character or a "white" part of an image is also regarded as a printable object. The length of **TESTFEED** and **FORMFEED** operations is decided by the *Media; Media Size; Length* setup.

The *Feedadjust* setup allows you to perform two global adjustments to the paper feed described above:

- *Start Adjust*
- *Stop Adjust*

By default, both these two parameters are set to 0, which allows for proper tear-off operation when there is no requirement of printing immediately at the forward edge of the label (or equivalent media).

- Start Adjust decides how much paper will be fed out or pulled back before the **FORMFEED**, **TESTFEED** or **PRINTFEED** is executed. Usually, there is a small distance between the dispenser shaft or tear off edge and the printhead. Thus, if you e.g. want to start printing directly at the forward edge of the label, you must pull back the paper before printing by means of a negative start adjust value.
- Stop Adjust decides how much extra or less paper will be fed out after the **FORMFEED**, **TESTFEED** or **PRINTFEED** is executed.

Note that so far we have only discussed how the paper feed will work regardless which program is run or what labels are printed. There are several ways to let the program control the paper feed without changing the setup:

FORMFEED
As already mentioned, if the **FORMFEED** statement is issued without any specification of the feed length, it will feed out a complete blank label (or the equivalent). But the **FORMFEED** statement can also specified as a positive or negative number of dots. However, it is **not** recommended to use this facility to substitute or modify the global Start Adjust and Stop Adjust setup as a part of the program execution.

LBLCOND
The **LBLCOND** statement can be used to override the values for the Start Adjust and/or Stop Adjust set in the Setup Mode. It can also be used to disable the LSS/BMS for a specified length of paper feed, e.g. to avoid text or pictures on the backside of a ticket being mistakenly detected as black marks, or when using irregularly shaped labels.

11.1 Paper Feed, cont'd.

The relation between paper and printhead when the **PRINTFEED** statement is executed decides all positioning along the Y-axis, i.e. along the paper web. Likewise, the relation between the paper and the cutting edge when a **CUT** statement is executed decides where the paper will be cut off.

11.2 Printing

The following instructions are used in connection with the actual printing:

CUT	Activates the optional paper cutter.
CUT ON/OFF	Enables/disables automatic cut-off operation in connection with each **PRINTFEED** statement.
LTS& ON/OFF	Enables/disables the label-taken sensor.
PRINT KEY ON/OFF	Enables/disables **PRINTFEED** execution by means of the Print key.
PRINTFEED	Prints a single label, ticket, tag or piece of strip, or a batch of labels, tickets etc.

CUT

Activates the optional paper cutter. As opposed to the **CUT ON/OFF** statement (see below), this statement allows you to control the cutter independently from the **PRINTFEED** statements. Since there is a longer distance from the printhead to the cutting edge than to the tear-off edge, the paper feed may need to be adjusted by means of the Start- and Stopadjust setup.

CUT ON/OFF

Enables/disables automatic cut-off initiated by each **PRINTFEED** statement and also allows you to decide the distance in dots by which the paper will be fed out before cutting and pulled back afterwards.

LTS& ON/OFF

These statements enables or disables the label-taken sensor, which is an photoelectrical sensor that detects when a label has not been removed from the printer's outfeed slot, and holds the printing until the label has been removed.

PRINT KEY ON|OFF

These two instructions can only be issued in the Immediate Mode and in the UBI Direct Protocol and enables/disables a single **PRINTFEED** operation to be automatically executed each time the <**Print**> key on the printer's built-in keyboard is pressed.

11.2 Printing, cont'd.

PRINTFEED (PF)

At the execution of a **PRINTFEED** statement, the firmware processes all previously entered text fields, bar code fields, image fields, box fields and line fields (see chapter 10) into a bitmap pattern. The bitmap pattern controls the heating of the printhead dots and the stepper motor that feeds the paper past the printhead. By default, each **PRINTFEED** statement produces one single copy, but the size of a batch of labels (or the equivalent) can optionally be specified.

After the execution of a **PRINTFEED** statement, the following statements are reset to their respective default values:

Statement	Default
ALIGN	1
BARFONT	"Swiss 721 BT", 12, 0, 6, 1, OFF
BARFONT ON/OFF	OFF
BARHEIGHT	100
BARMAG	2
BARRATIO	3, 1
BARSET	"INT2OF5", 3, 1, 2, 100, 2, 1, 2, 0, 0
BARTYPE	"INT2OF5"
DIR	1
FONT	"Swiss 721 BT", 12, 0
INVIMAGE	NORIMAGE
MAG	1, 1
PRPOS	0, 0

This does only affect new statements executed after the **PRINTFEED** statement, but not already executed statements. The amount of paper fed out at the execution of a **PRINTFEED** statements under various conditions is discussed in chapter 11.1.

Example (printing identical labels):

```
10    PRPOS 100, 100
20    FONT "Swiss 721 Bold BT", 14, 10
30    PRTXT "TEST LABEL"
40    PRINTFEED 5
RUN
```

Example (printing five copies of the same label layout with consecutive numbering):

```
10    FOR A%=1 TO 5
20    PRPOS 100, 100
30    FONT "Swiss 721 Bold BT", 14, 10
40    PRTXT "LABEL ";A%
50    PRINTFEED
60    NEXT A%
RUN
```

11.3 Length of Last Feed Operation

ACTLEN
This function returns the approximate length in dots of most recently executed paper feed operation. It can for example be used to determine the length of the labels before printing a list, so the list can be divided into portions that fit the labels.

Example:

```
10    FORMFEED
20    PRINT ACTLEN
RUN
```

11.4 Batch Printing

The term “Batch Printing” means the process of printing several labels without stopping the paper feed motor between labels. The labels may be exact copies or differ more or less in appearance.

When a **PRINTFEED** is executed, the firmware processes the program instructions into a bitmap pattern and stores it in one of the two image buffers in the printer's temporary memory. The image buffer compensates for differences between processing time and printing time.

Next step is to use the bitmap pattern to control the heating of the printhead dots while the ribbon and/or paper is fed past the printhead. Obviously, the print speed causes the image buffer to be emptied more quickly.

Normally, when the first image buffer is emptied and the printing is completed, the firmware can process a new bitmap pattern and store it in the second image buffer. By means of an **OPTIMIZE "BATCH" ON** statement, you can make the firmware start processing next label image and store it in the second image buffer while the first label is still in process of being printed. Thus, by switching between the two image buffers, a high continous print speed can be maintained.

There are a number of instructions that facilitate batch printing:

FIELDNO	Divides the program into portions that can be cleared individually.
CLL	Clears part or all of the image buffer.
OPTIMIZE "BATCH" ON	Enables optimizing.
OPTIMIZE "BATCH" OFF	Disables optimizing.

When using batch printing, consider this:

- The program must be written as to allow batch printing.
- In case of small differences between labels, make use of **CLL** and **FIELDNO** instructions and write the program so the variable data are processed last.
- Always use the **OPTIMIZE "BATCH" ON** strategy.

Should a the printer stop between labels, lower the print speed somewhat. Usually, the overall time to produce a certain number of labels is more important than the actual print speed.

11.4 Batch Printing, cont'd.

CLL & FIELDNO

The image buffer stores the bitmap pattern of the label layout between processing and printing. The image buffer can be cleared partially or completely by means of a **CLL** statement.

- Complete clearing is obtained by a **CLL** statement without any reference to a field (see below) and is useful to avoid printing a faulty label after certain errors have occurred.
- Partial clearing is used in connection with print repetition when only part of the label should be modified between the copies. In this case, the **CLL** statement must include a reference to a field, specified by a **FIELDNO** function. When a **CLL** statement is executed, the image buffer will be cleared from the specified field to the end of the program.

In this example, the text "Month" is kept in the image buffer, whereas the names of the months are cleared from the image buffer as soon as they are printed, one after the other:

```
10    FONT "Swiss 721 Bold BT",18,10
20    PRPOS 100,300
30    PRTXT "MONTH:"
40    PRPOS 100,200
50    A%=FIELDNO
60    PRTXT "JANUARY":PRINTFEED
70    CLL A%
80    FONT "Swiss 721 Bold BT",18,10
90    PRPOS 100,200
100   PRTXT "FEBRUARY":PRINTFEED
110   CLL A%
120   FONT "Swiss 721 Bold BT",18,10
130   PRPOS 100,200
140   PRTXT "MARCH":PRINTFEED
150   CLL A%
RUN
```

OPTIMIZE "BATCH" ON/OFF

This statement is used to speed up batch printing. The program execution will not wait for the printing of a label to be completed, but proceeds executing next label image into the other image buffer as soon as possible.

The default setting is **OPTIMIZE "BATCH" OFF**. However, if all the following conditions are fulfilled, **OPTIMIZE "BATCH" ON** will automatically be invoked:

- A value >1 is entered for the **PRINTFEED** statement.
- **LTS& OFF** *(default)*
- **CUT OFF** *(default)*

OPTIMIZE "BATCH" ON revokes **OPTIMIZE "BATCH" OFF**.

12. Fonts

12.1 Font Types

UBI Fingerprint 7.xx supports scalable single- and double-byte fonts in TrueDoc (.PFR = Portable Font Resource) and TrueType (.TTF) format that comply with the Unicode standard.

TrueDoc fonts in .PFR format can only be obtained from UBI. A single .PFR file can contain a number of different fonts. Compared with TrueType fonts, TrueDoc fonts require less memory spaces and work faster.

☞ ***Standard Fonts***
Also see:
• Chapter 12.5
• UBI Fingerprint 7.11 Reference Manual

UBI Fingerprint 7.11 contains 15 single-byte standard fonts in the systems parts ("Kernel") of the permanent memory (device "rom:).

TrueType fonts from sources other the UBI could normally be used provided they comply with the Unicode standard. This is usually the case with TrueType fonts for Windows 95 and Windows NT.

12.2 Single-byte Fonts

Single-byte fonts are fonts that are mapped in the range of ASCII 0-127 dec (7-bit communication) or ASCII 0-255 dec (8 bit communication). Example of single-byte fonts are Latin, Greek, Cyrillic, Arabic and Hebrew fonts.

Single-byte fonts are selected by means of the statements **FONT** and **BARFONT** (see chapter 10.2 and 10.3 respectively) and the corresponding character set by means of the statement **NASC** (see chapter 9.1).

12.3 Double-byte Fonts

☞ ***Unicode***
Also see:
• http://www.unicode.org

Double-byte fonts are fonts that are mapped in the area of ASCII 0-65,536 dec. 8 bit communication must be selected. This means that any glyph (i.e. characters, interpunctation marks, symbols, digits etc.) in the Unicode World Wide Character Standard, can be specified. In its current version (2.0), Unicode contains 38,885 glyphs. Example of languages that require double-byte fonts are Chinese, Japanese and Korean.

Double-byte fonts are selected by means of the statement **FONTD** (see chapter 10.2) and the corresponding character set by means of the statement **NASCD** (see chapter 9.1). Note that double-byte fonts **cannot** be used for bar code interpretations (**BARFONT**).

12.4 Font Direction, Size and Slant

Fonts can be rotated in 4 directions using a **DIR** statement. Using the **FONT**, **FONTD** and **BARFONT** statements, fonts can be specified in regard of size in points (1 point = 1/72" = 0.352 mm) and slant in degrees (clockwise). It is also possible to magnify fonts using a **MAG** statement. This facility is mainly retained for compatibility with earlier UBI Fingerprint versions, since the printout quality will suffer. We recommend specifying a larger size in points rather than using a **MAG** statement.

12.5 Standard Fonts

☞ ***Font Printout Samples***
See:
• UBI Fingerprint 7.11 Reference Manual

As standard, the UBI Fingerprint firmware contains 15 single-byte TrueDoc fonts stored in the systems part ("Kernel") of the permanent memory. In the **FONT** and **BARFONT** statements, the full names according to the list below must be used (case sensitive).

- Century Schoolbook BT
- Dutch 801 Roman BT
- Dutch 801 Bold BT
- Futura Light BT
- Letter Gothic 12 Pitch BT
- Monospace 821 BT
- Monospace 821 Bold BT
- OCR-A BT *(see note)*
- OCR-B 10 Pitch BT *(see note)*
- Prestige 12 Pitch Bold BT
- Swiss 721 BT
- Swiss 721 Bold BT
- Swiss 721 Bold Condensed BT
- Zapf Dingbats BT *(see note)*
- Zurich Extra Condensed BT

Note:
When selecting OCR-A BT, OCR-B 10 Pitch BT or Zapf Dingbats BT, the printer will automatically switch from the presently selected character set to a special one for the font in question (see later in this chapter). As soon as any other font is selected again, the printer will automatically return to the previously selected character set.

12.6 Old Font Names

To maintain compatibility with earlier versions of UBI Fingerprint, the old font name convention for naming standard bitmap fonts can also be used, e.g. "SW030RSN" or "MS060BMN.2". The firmware will select the corresponding TrueDoc font in the printer's memory and set its parameters so its appearance and size come as close to the specified bitmap font as possible.

12.7 Adding Fonts

The standard complement of fonts listed in chapter 12.5 can be supplemented by additional fonts using three different methods:

- ***Downloading fonts from a Font Install Card.***
  The card must be inserted before the printer is started. At startup the fonts are automatically downloaded, installed and permanently stored in the printer's memory. The fonts can be used without the card being present
- ***Using fonts from a Font Card.***
  The card must be inserted before the printer is started. At startup the fonts are automatically installed, but not copied to the printer's memory (i.e. the card must always be present before such a font can be used).
- ***Downloading font files.***
  Font files can be downloaded and installed by means of either of the two statements **IMAGE LOAD** and **TRANSFER KERMIT**. There is no need to restart the printer before using the font in question.

Note:
Double-byte fonts are often too large to be stored in the printer's memory. In such cases, a Font Card must be used.

12.8 Listing Fonts

Regardless in which parts of the memory the different fonts are stored, they can all be listed to the standard OUT channel by a single statement, namely **FONTS**. This statement does not list dedicated bar code fonts.

Another method of listing fonts is to use a **FONTNAME$** function, which also will list dedicated barcode fonts.

Font files can be listed to the standard OUT channel by means of the **FILES** statement.

This example shows how all fonts can be listed:

```
10    A$ = FONTNAME$(0)
20    IF A$ ="" THEN END
30    PRINT A$
40    A$ = FONTNAME$(-1)
50    GOTO 20
RUN
```

12.9 Removing Fonts

Font files stored in the read/write devices ("c:", "tmp:" and "card1:") can be deleted using **KILL** statements. Even if a font file is **KILLED**, the name of the font will still be listed e.g. by a **FONTS** statement until the printer is restarted. Note that the names of the font files may differ from the name of the font.

12.10 Font Aliases

The names of the standard fonts UBI Fingerprint are rather long and may be cumbersome to use. They are also incompatible with the **LAYOUT** statement, which restricts the font and barfont names to 10 characters.

However, it is possible to create a file containing a list of font aliases. The file should be named exactly as shown here (note the leading period character that specifies it as a system file):

```
"c:.FONTALIAS"
```

The format of the file should be:

"<Alias name #1>","<Name of font>"[,size[,<slant>]]

"<Alias name #2>","<Name of font>"[,size[,<slant>]]

"<Alias name #3>","<Name of font>"[,size[,<slant>]]

etc., etc.

The file can contain as many fontname aliases as required. The default size is 12 points and the default slant is 0°.

A font alias can be used as any other font, but its size and slant can not be changed.

Examples:

```
"BODYTEXT","Century Schoolbook BT",10
"HEADLINE","Swiss 721 Bold BT",18
"WARNING","Swiss 721 BT",12,10
```

13. Bar Codes

13.1 Standard Bar Codes

A large number of commonly used bar code symbologies are included in the systems part (“Kernel”) of the printer's permanent memory.

Some bar codes require special barcode fonts, e.g. UPC and EAN bar codes.

Bar codes cannot be listed by means of any UBI Fingerprint instruction. As standard, UBI Fingerprint 7.11 contains the following bar codes.

Bar Code Type	Designation
Codabar	"CODABAR"
Code 11	"CODE11"
Code 39	"CODE39"
Code 39 full ASCII	"CODE39A"
Code 39 w. checksum	"CODE39C"
Code 93	"CODE93"
Code 128	"CODE128"
DUN-14/16	"DUN"
EAN-8	"EAN8"
EAN-13	"EAN13"
EAN-128	"EAN128"
Five-Character Supplemental Code	"ADDON5"
Industrial 2 of 5	"C2OF5IND"
Industrial 2 of 5 w. checksum	"C2OF5INDC"
Interleaved 2 of 5	"INT2OF5"
Interleaved 2 of 5 w. checksum	"I2OF5C"
Interleaved 2 of 5 A	"I2OF5A"
LEB	"LEB"
Matrix 2 of 5	"C2OF5MAT"
MSI (modified Plessey)	"MSI"
Plessey	"PLESSEY"
PDF 417	"PDF417"
Philips	"PHILIPS"
Philips (alternative designation)	"DOT CODE A"
Plessey	"PLESSEY"
Straight 2 of 5	"C2OF5"
Two-Character Supplemental Code	"ADDON2"
UCC-128 Serial Shipping Container Code	"UCC128"
UPC-5 digits Add-On Code	"SCCADDON"
UPC-A	"UPCA"
UPC-D1	"UPCD1"
UPC-D2	"UPCD2"
UPC-D3	"UPCD3"
UPC-D4	"UPCD4"
UPC-D5	"UPCD5"
UPC-E	"UPCE"
UPC Shipping Container Code	"UPCSCC"
USD5	"USD5"

13.2 Setup Bar Codes

Some printers can be set up by means of special Code 128 bar codes that are read using a Bar Code Wand. Refer to UBI Fingerprint 7.xx Reference Manual.

14. IMAGES

14.1 Images vs Images Files

There is a distinction between “Images” and “Image Files”:

- “Image” is a generic term for all kinds of printable pictures, e.g. symbols, logotypes or other illustrations, in the internal bitmap format of UBI Fingerprint.
- “Image Files” are files in various bitmap formats that can be converted to “Images” in the internal bitmap format of UBI Fingerprint. Images files can be stored in the printer's memory, but cannot be used for printing before they have been converted to “Images”.

14.2 Standard Images

As standard, the systems part (“Kernel”) of the printer's permanent memory contains a number of images primarily used for printing test labels.

14.3 Downloading Image Files

☞ *Downloading via Kermit*
Also see:
• Chapter 6.8

Image files in .PCX format can be downloaded to the printer using the Kermit protocol and then converted to UBI's internal image format by means of the instruction **RUN "pcx2bmp"** (see chapter 6.5) .

Image files in .PCX format can also be both downloaded, automatically converted to images and installed by means of the **IMAGE LOAD** statement.

☞ *Image Transfer Protocols*
Also see:
• UBI Fingerprint 7.11 Reference Manual

Image files in Intel hex formats, or formats according to UBI Fingerprint file transfer protocols **UBI00**, **UBI01**, **UBI02**, **UBI03**, or **UBI10**, can be downloaded to the printer using the instructions **STORE IMAGE**, **STORE INPUT** and **STORE OFF**, e.g.:

```
10    STORE OFF
20    INPUT "Name:", N$
30    INPUT "Width:", W%
40    INPUT "Height:", H%
50    INPUT "Protocol:", P$
60    STORE IMAGE N$, W%, H%, P$
70    STORE INPUT 100
80    STORE OFF
RUN
```

The system variable **SYSVAR** allows you to check the result of an image download by means of **STORE INPUT**:

- **SYSVAR (16)** reads the number of bytes received.
- **SYSVAR (17)** reads the number of frames received.

Both values are reset when a new **STORE IMAGE** statement is executed.

14.4 Listing Images

The names of all images stored in the various parts of the printer's memory can be listed to the std. OUT channel by means of an **IMAGES** statement or a program using the **IMAGENAME$** function.

Image files can be listed to the std. OUT channel by means of a **FILES** statement.

Example:
This example lists all images the the printer's memory (in this case only standard images):

```
IMAGES
```

yields:

```
CHESS2X2.1                CHESS4X4.1
DIAMONDS.1                UBI.1
UBI.2                     UBI010.1
UBI010.2

1543536 bytes free   307456 bytes used
Ok
```

14.5 Removing Images

Images can be removed from the read/write devices (i.e. "c:", "tmp:" and "card1:") using **REMOVE IMAGE** statements.

Images files can be removed from the read/write devices (i.e. "c:", "tmp:" and "card1:") using a **KILL** statement.

15. Printer Function Control

15.1 Keyboard

Note:
External keyboard do not work in the Setup Mode.

All UBI Fingerprint 7.xx-compatible printers are provided with a built-in keuboard containing a set of numeric keys supplemented with a number of function keys. Separate alphanumeric keyboards are available as options.

The keys have three purposes:
- To control the printer in the Setup Mode, and to some extent also in the Immediate Mode.
- To enter input data in the form of ASCII characters.
- To make the program execution branch to subroutines according to **ON KEY . . . GOSUB** statements.

Note that input from the printer's keyboard (see chapter 7.6) excludes the use of **ON KEY . . . GOSUB** statements (see chapter 5.8) and vice versa.

Controlling the Printer in the Setup and Immediate Modes:
- The use of the keyboard in the Setup Mode is described in the Installation & Operation manual for the printer model in question.
- In a printer running in the Immediate Mode, only four keys are working:
  - The <**Print**> key or button produces a **FORMFEED** operation, or – if the printhead is lifted – runs the printer's print roller a number of rotations in order to facilitate cleaning (**CLEANFEED**).
  - The <**Feed**> key works the same way as the <**Print**> button.
  - The <**Shift**> + <**Print**> keys pressed simultaneously produce a **TESTFEED** operation.
  - The <**Setup**> key gives access to the Setup Mode.
- In the Immediate Mode, the printing of labels by means of the print key can be enabled or disabled using a **PRINT KEY ON/OFF** statement, also see chapter 11.3.

Enabling the Keys
Before a key can be used to make the execution branch to a subroutine using an **ON KEY . . . GOSUB** statement, the key must be enabled using a **KEY . . . ON** statement. Enabled keys can also be disabled again using **KEY . . . OFF** statements.

However, the keyboard can also be used to enter input data (provided "console:" is **OPENed** for **INPUT**), and also be used in the Setup and Test Modes, regardless if the keys are enabled or not.

15.1 Keyboard, cont'd.

Key Id. Numbers

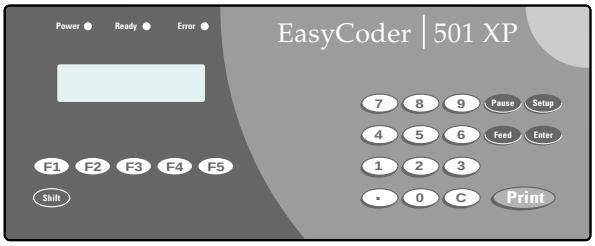
The keys are specified by id. numbers in connections with the following statements:

`KEY...ON`	Enables the specified key.
`KEY...OFF`	Disables the specified key.
`ON KEY...GOSUB...`	Branches the program execution to a sub-routine when the specified key is pressed.

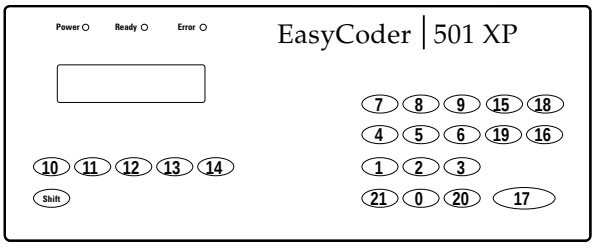
Each key has two id. numbers, one for its **unshifted** position and another for its **shifted** position[1]. To select the shifted position of a certain key, keep the <**Shift**> key depressed while you press the desired key. The id. number of the shifted key is equal to its unshifted id. number + 100. For example, the <**F1**> key has id. number 10 in unshifted position, but id. number 110 in shifted position.

The illustration below shows the default id. numbers of the keyboard of the EasyCoder 501 XP. The id. number of the <**Print**> button or key also applies to printers models without keyboard.

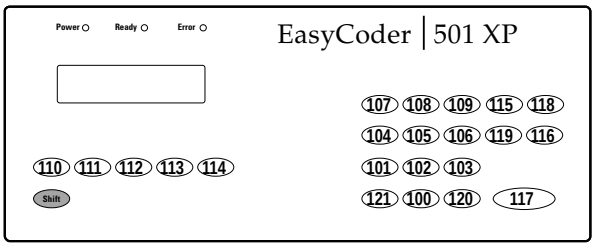
If the keyboard is remapped (see later in this chapter), the id. numbers will be affected.



Actual keyboard apperance



Unshifted keys i.d. numbers



Shifted keys i.d. numbers

15.1 Keyboard, cont'd.

Key-initiated Branching

What will happen when an enabled key is pressed can be decided by an **ON KEY...GOSUB** statement, that branches the program execution to a subroutine, where additional instructions specify the action to be taken. Refer to chapter 5.8 for further information and additional program example.

*Here is an example of how two keys (<**F1**> and <**F2**>) are enabled and used to branch to different subroutines. The keys are specified by their id. numbers (10 and 11 respectively):*

```
10    KEY (10) ON: KEY (11) ON
20    ON KEY (10) GOSUB 1000
30    ON KEY (11) GOSUB 2000
40    GOTO 40
50    END
1000  PRINT "You have pressed F1"
1010  RETURN 50
2000  PRINT "You have pressed F2"
2010  RETURN 50
RUN
```

Audible Key Response

Each time a key is pressed, the printer's beeper will, by default, emit a short signal (1200 Hz for 0.03 sec). The frequency and duration of the signal can be globally changed for all keys by means of a **KEY BEEP** statement. Obviously, setting the frequency and/or duration to 0 will turn off the signal for all keys.

Input from Printer's Keyboard:

Provided "console:" is **OPENed** for sequential **INPUT**, the keys can be used to enter ASCII characters to the program using the following instructions:

INPUT#	reads a string of data to a variable.
INPUT$	reads a limited number of characters to a variable.
LINE INPUT#	reads an entire line to a variable.

Refer to chapter 7.6 for a table showing the ASCII values that the various keys generate and for a program example. Note that input from keyboard does not require any keys to be enabled.

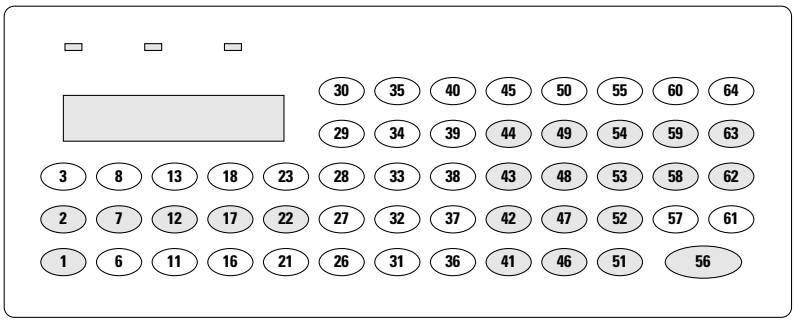
15.1 Keyboard, cont'd.

Remapping the Keyboard

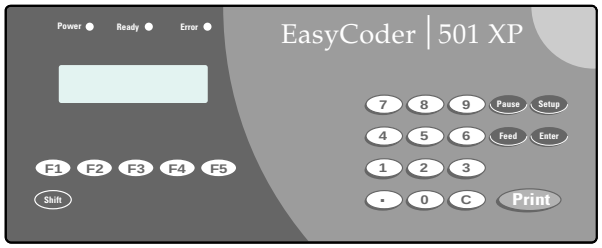
The keyboards of the various printer models are fully remappable (with exception for the <**Shift**> key), as to allow the printer to be adapted to special applications or national standards using the instruction **KEYBMAP$**. Thus you can decide which two ASCII characters each key will produce, with and without the Shift key being activated. The mapping also decides the id. numbers for the keys.

The basis of the remapping process is the position number of each key, as illustrated for the EasyCoder 501 XP below. Note that in the Setup Mode, the keys have fixed positions that are not affected by any **KEYBMAP$** instructions.

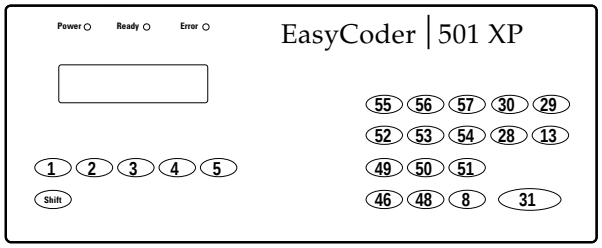
Note the distinction between id. numbers and position numbers!



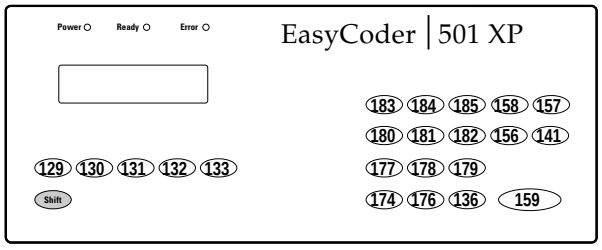
Keyboard position numbers on an EasyCoder 501 XP. The keys printed on the keyboard overlay are marked with a shade of grey. Key positions 1 and 30 cannot be remapped.



Actual keyboard apperance



Unshifted keys ASCII values



Shifted keys ASCII values

15.1 Keyboard, cont'd.

Remapping the Keyboard, cont'd.

The present keyboard mapping can be read to a string variable using the **KEYBMAP$** instruction with the following syntax:

<string variable>=KEYBMAP$(n) *where....*

n = 0 reads the unshifted characters.

n = 1 reads the shifted characters.

This example reads the unshifted characters on the keyboard of an EasyCoder 501 XP. Non-existing key positions get ASCII value 0:

```
10    PRINT "Pos","ASCII","Char."
20    A$=KEYBMAP$(0)
30    FOR B%=1 TO 64
40    C$=MID$(A$,B%,1)
50    E%=ASC(C$)
60    PRINT B%,E%,C$
70    NEXT
RUN
```

You can also use the **KEYBMAP$** instruction to remap the keyboard, using the following syntax:

KEYBMAP$(n) = <string> *where...*

n = 0 maps the unshifted characters in ascending position number order.

n = 1 maps the shifted characters in ascending position number order.

The string that contains the desired keyboard map should contain the desired character for each of 64 key positions (in ascending order) regardless if the keyboard contains that many keys.

Characters, that cannot be produced by the keyboard of the host, can be substituted by **CHR$** functions, where the character is specified by its ASCII decimal value according to the selected character set (see **NASC** statement). The same applies to special characters. See table below.

Non-existing key positions are mapped as Null, i.e. **CHR$(0)**.

ASCII decimal values for Special Keys

Key	Unshifted	Shifted
F1	1	129
F2	2	130
F3	3	131
F4	4	132
F5	5	133
Pause	30	158
Setup	29	157
Feed	28	156
Enter	13	141
C (Clear)	8	136
Print	31	159

15.1 Keyboard, cont'd.

Remapping the Keyboard, cont'd.

*The following example reads back the current keyboard map and changes the <**F1**> key to A, the <**F2**> key to B, and the <**F3**> key to C:*

```
10    A$=KEBMAP$(0)
20    B$=LEFT$(A$,1)+"A"+MID$(A$,3,4)+"B"+
      MID$(A$,8,4)+"C"+MID$(A$,13)
30    KEYBMAP$(0)=B$
RUN
```

The following example illustrates the mapping of the keyboard for an EasyCoder 501 XP (unshifted keys only). Note the limit of max. 300 characters per program line:

```
10    B$=CHR$128+CHR$1+STRING$(4,0)+CHR$(2)+
      STRING$(4,0)+CHR$(3)
20    B$=B$+STRING$(4,0)+CHR$(4)+STRING$(4,0)+
      CHR$(5)+STRING$(18,0)
30    B$=B$+".147"+CHR$(0)+"0258"+CHR$(0)+
      CHR$(8)+"369"+CHR$(0)+CHR$(31)
40    B$=CHR$(0)+CHR$(28)+CHR$(30)+STRING$(2,0)+
      CHR$(13)+CHR$(29)+CHR$(0)
50    KEYBMAP$(0)=B$
RUN
```

15.2 Display

All present UBI Fingerprint 7.xx-compatible printers have a 2 × 16 characters LCD (Liquid Crystal Display). The UBI Fingerprint firmware uses it to show a number of standardized messages, e.g. in the Setup Mode, but it can also be controlled by programming instructions (see *"Output to Display"* below). The display is provided with a controllable cursor, as described later in this chapter (*"Cursor Control"*).

Output to Display

Before you can print any text to the display, it must be opened for sequential output, e.g.:

```
10    OPEN "console:" FOR OUTPUT AS 1
```

Then you should clear any previously displayed message by sending two empty **PRINT#** or **PRINTONE#** statements:

```
20    PRINT#1:PRINT#1
```

☞ ***Clearing the Display***
Also see:
• "Cursor Control: Clearing the Display" later in this chapter.

Now you can send a string to each of the two lines. Note the appending semicolon on the second line:

```
30    PRINT#1, "Upper line"
40    PRINT#1, "Lower line";
RUN
```

This will result in the following message being displayed:

```
Upper line
Lower line
```

As a alternative to sending two separate lines, you can also send a single line consisting of max. 33 characters, where:

- Character 1–16 specifies the upper line
- Character No. 17 is not displayed at all
- Character No. 18–33 specifies the lower line
- The line should be appended by a semicolon (;).

Using this method, the example above would look like this (underscore characters indicate space characters):

```
10    OPEN "console:" FOR OUTPUT AS 1
20    PRINT#1:PRINT#1
30    PRINT#1,"Upper_line_____Lower_line";
RUN
```

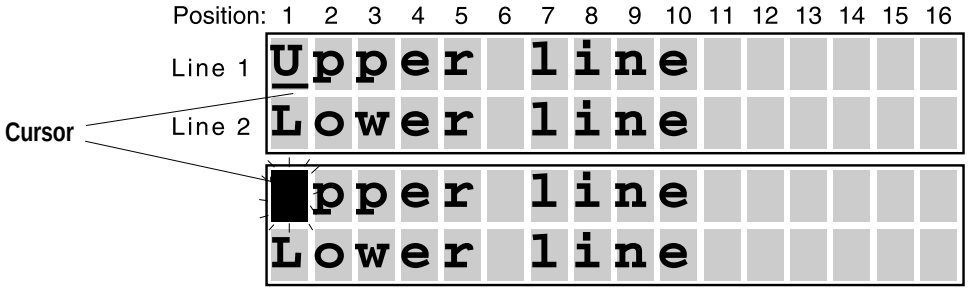
15.2 Display, cont'd.

Cursor Control

The cursor control instructions can be used for four purposes:

- To clear the display from messages (as an alternative to the double **PRINT#** statement on line 20 in the example above).
- To enable or disable the cursor.
- To select cursor type (underscore or block/blink)
- To place the cursor at a specified position or to move it.

The cursor is either a black line under a character position in the display, or a blinking block that intermittently blacks out the character position:



Each cursor control command should start with the character CSI (Control Sequence Introducer) = ASCII 155 decimal, or (in case of 7-bit communication) with the characters "**ESC**" + "**[**" (ASCII 27 + 91 decimal).

Clearing the Display:

Syntax: <CSI> + <<0|1|2>J> *where:*

CSI	=	ASCII 155 dec.
0	=	From active position to end, inclusive (default)
1	=	From start to active position, inclusive
2	=	All of the display
J	=	Must always append the string

Example (clears all of the display):

```
10    OPEN "console:" FOR OUTPUT AS 1
20    PRINT#1, CHR$(155) + "2J";
```

Selecting Cursor Type:

Syntax: <CSI> + <4p|5p> *where:*

CSI	=	ASCII 155 dec.
4p	=	Underscore
5p	=	Block/Blink (default)

Example (selects underscore-type cursor):

```
10    OPEN "console:" FOR OUTPUT AS 1
20    PRINT#1, CHR$(155) + "4p";
```

15.2 Display, cont'd.

Cursor Control, cont'd.:

Enabling/Disabling the Cursor:

Syntax: <CSI> + <2p|3p> *where:*

CSI = ASCII 155 dec.
2p = Cursor On
3p = Cursor Off (default)

Example (enables the cursor):

```
10    OPEN "console:" FOR OUTPUT AS 1
20    PRINT#1, CHR$(155) + "2p";
```

*Note that a semicolon should append the **PRINT#** instructions in order to avoid interfering with existing messages in the display.*

Setting the Absolute Cursor Position:

Syntax: <CSI> + <<v>;<h>H> *where:*

CSI = ASCII 155 dec.
v = Is the line (1 = Upper; 2 = Lower)
h = Is the position in the line (1–16)
H = Must always append the string
If **v**, **h** or both are missing, the default value is 1.

Example (setting the cursor in upper left position):

```
10    OPEN "console:" FOR OUTPUT AS 1
20    PRINT#1, CHR$(155) + "H";
```

Example (setting the cursor in lower right position):

```
10    OPEN "console:" FOR OUTPUT AS 1
20    PRINT#1, CHR$(155) + "2;16H";
```

Move the Cursor Relative Current Position:

Syntax: <CSI><n>A|B|C|D *where:*

CSI = ASCII 155 dec.
n = Is number of steps relative current position (default 1)
A = Is direction Up
B = Is direction Down
C = Is direction Forward
D = Is direction Backward

The relative movement must not place the cursor outside the display area (2 × 16 positions) or the instruction will be ignored.

Example (moving the cursor from the first position in the upper line to the last position in the lower line):

```
10    OPEN "console:" FOR OUTPUT AS 1
20    PRINT#1, CHR$(155) + "1B";
30    PRINT#1, CHR$(155) + "15C";
```

15.3 LED Control Lamps

Beside showing messages in the printer's display window (see chapter 15.2, the program can use two of the three LED's (Light Emitting Diodes) on the printer's front panel to notify the operator of various conditions.

There are two statements for control the LED's:

LED...ON	Turns the specified LED on.
LED...OFF	Turns the specified LED off.

The printer's front panel contains three LED's labelled ”Power”, ”Ready” (0), and ”Error” (1):

- The “Power” LED is connected to the printer's power supply and is lit when the power is on. It cannot be controlled by the program.
- The two other LED's (“Ready” and “Error”) can be programmed at will using **LED...ON** and **LED...OFF** statements, even though the printed text on the keyboard imposes certain restrictions.

Example:
In this example, the “Ready” LED (0) is lit until an error occur. Then the “Error” LED (1) is lit instead. The “Error” LED remains lit until the error is cleared. A suitable error can be generated by running the program with the printhead lifted.

```
10 LED 0 ON
20 LED 1 OFF
30 ON ERROR GOTO 1000
40 PRPOS 100,100
50 FONT "Swiss 721 Bold BT",36
60 PRTXT "OK!"
70 PRINTFEED
80 LED 0 ON
90 LED 1 OFF
100 END
1000 LED 0 OFF
1010 LED 1 ON
1020 RESUME
RUN
```

15.4 Buzzer

In addition to the visual signals given by means of the display and the LED control lamps (see chapter 15.2 and 15.3), audible signals can also be initiated by the program execution in order to notify the operator.

The following instructions can be used:

BEEP	Initiates a short signal of fixed frequency and duration.
SOUND	Initiates a signal vith variable frequency and duration.

The buzzer can be controlled by either a **BEEP** statement, which gives a short shrill signal (≈800 Hz for 0.25 sec.), or by a **SOUND** statement, which allows you to vary both the frequency and duration. You can even compose your own melodies, if your musical ear is not too sensitive!

In this example, a warning signal is emitted from the buzzer e.g. when the error "printhead lifted" occurs and keeps sounding until the error is cleared. A short beep indicates that the printer is OK.

```
10 ON ERROR GOTO 1000
20 PRPOS 100,100
30 FONT "Swiss 721 Bold BT", 36
40 PRTXT "OK!"
50 PRINTFEED : BEEP
60 END
1000 SOUND 880,25 : SOUND 988,25 : SOUND 30000,10
1010 RESUME
RUN
```

15.5 Clock/Calendar

The UBI Fingerprint 7.xx-compatible printers are fitted with a real-time clock circuit (RTC). The RTC is battery backed-up and will keep running even when the printer is turned off.

Please refer to chapter 9.3 for information on how to read the printer's clock/calendar, and on the standard formats for date and time.

The following instructions are to set the clock/calendar:

DATE$ = <sexp>	Sets the date (YYMMDD format)
TIME$ = <sexp>	Sets the time (HHMMSS format)

Example (setting the clock/calendar to 08.11.30 January 23, 1998):

```
DATE$ = "980123"
TIME$ = "081130"
```

Note that the values must always be entered as string expressions. Possible numeric expressions can be converted to string format using **STR$** functions (see chapter 9.2).

15.6 Printer Setup

> ☞ ***UBI Shell Startup Program***
> Also see:
> • Installation & Operation manual

The printer's setup can be changed manually in the Setup Mode using the built-in keyboard[1] or remotely be means of the Terminal Setup in UBI Shell.

Detailed information on the methods of manual or terminal setup and the various setup parameters can be found in the Installation & Operation manual for the printer model in question.

If you want to change some setup parameter either by remote control (other than Terminal Setup) or as a part of the program execution, you can use the **SETUP** statement.

SETUP

This statement can be used in four different ways:

SETUP	Makes the printer enter the Setup Mode.
SETUP WRITE	Creates a copy of the printer's current setup and saves it as a file in the printer's memory under a specified name or returns the current setup to the specified communication channel.
SETUP<file name>	Changes some or all of the setup parameters in the printer's current setup according to a setup file.
SETUP<string>	Changes a single setup parameter.

Reading the Current Setup

The easiest way to read the printer's current setup is to use a **SETUP WRITE** statement to return the setup to the serial communication channel used for output to the host (usually "uart1:").

Example:

```
SETUP WRITE "uart1:"
```

Creating a Setup File

Create a setup file using UBI Fingerprint instructions like this:

- **OPEN** a file for sequential **OUTPUT**. See chapter 8.3.
- Use a **PRINT#** statement to enter each parameters you want so change. The input must follow the stipulated syntax exactly (see the UBI Fingerprint 7.xx Reference Manual, **SETUP** statement).
- **CLOSE** the file.

[1]/. An external keyboard cannot be used in the Setup Mode.

15.6 Printer Setup, cont'd.

Changing the Setup using a Setup File

Use a **SETUP<filename>** statement to change the printer's setup. If the setup file is stored in another part of the printer's memory than the current directory, the file name should contain a reference to the device in question.

In the following example, we will first save the current setup under a new file name and then make a setup file that changes the size of the transmit buffer on "uart1:" just a little. Finally, we use the setup file to change the printer's setup.

```
10    SETUP WRITE "SETUP1.SYS"
20    OPEN "SETUPTEST.SYS" FOR OUTPUT AS #1
30    PRINT#1,"SER-COM,UART1,TRANS BUF,310"
40    CLOSE #1
50    SETUP "SETUPTEST.SYS"
RUN
```

Changing the Setup using a Setup String

A single setup parameter can be changed without creating any file. The **SETUP** statement should be followed by a string following exactly the same syntax as the corresponding parameter in a Setup file, but without any leading **PRINT#** statement.

The same change as in the example above would look this way when using a setup string:

```
SETUP "SER-COM,UART1,TRANS BUF,310"
```

15.7 System Variables

Some sensors and other conditions can be read or set by means of the **SYSVAR** system variable.

SYSVAR

The following **SYSVAR** parameters are released for public use:

SYSVAR(13)	returns the value of the ribbon counter *(requires an optional sensor)*.
SYSVAR(14)	returns the number of errors since last power on.
SYSVAR(15)	returns the number of errors since the previously executed **SYSVAR(15)** instruction.
SYSVAR(16)	returns the number of bytes received at the execution of a **STORE** or **STORE INPUT** statement.
SYSVAR(17)	returns the number of frames received at the execution of a **STORE** or **STORE INPUT** statement.
SYSVAR(18)	returns or sets the verbosity level.
SYSVAR(19)	returns or sets the type of error messages transmitted by the printer.
SYSVAR(20)	returns 0 if the printer is set up for direct thermal or 1 if set up for thermal transfer printing.
SYSVAR(21)	returns the printhead density in dots/mm.
SYSVAR(22)	returns the number of dots in the printhead.
SYSVAR(23)	returns 1 if a transfer ribbon is detected, else 0.
SYSVAR(24)	returns 1 if a power-up has been performed since last **SYSVAR(24)**, else 0.
SYSVAR(25)	returns or selects the type of Centronics communication on the parallel communication port "centronics:": **SYSVAR(25)=0** Standard type **SYSVAR(25)=1** IBM/Epson type **SYSVAR(25)=1** Classic type
SYSVAR(28)	decides if the information on the position of the paper vs the printhead should be cleared or not when the printhead is lifted.
SYSVAR(32)	returns the odometer value, i.e. the length of paper that have been fed past the printhead in kilometres.

- Parameter 13 is intended for use with the optional ribbon low sensor kit.
- Parameters 14 and 15 are primarily intended for service purposes.
- Parameters 16 and 17 are used in connection with transfer of images from the host to the printer and are explained in chapter 14.3.

15.7 System Variables, cont'd.

- Parameter 18 is used for returning or setting the printer's verbosity level, i.e. the printer's response to received instructions as explained in chapter 7.7.
- Parameter 19 is used for returning or selecting one of four types of error messages, see chapter 16.1.
- Parameter 20 checks if the printer is printer is set up for direct thermal printing or thermal transfer printing, which depends on the choice of paper type in the Setup Mode, see the Technical Manual.
- Parameters 21 and 22 are used to check the printhead in regard of printhead density and number of dots respectively. Together with parameter 20 and the **VERSION$** function, see chapter 15.11, these parameters allows the program to identify different printer models. Thereby it is possible to design programs that will work in all EasyCoder printers.
- Parameter 23 checks the status of the ribbon end sensor in thermal transfer printers.
- Parameter 24 is useful, when certain data, e.g. date and time formats, are not generated as a part of the program execution. Since such data are stored in the temporary memory, they will be lost at power-up or reboot. Using **SYSVAR(24)**, the printer can be polled for power-ups, so lost data can be renewed.
- Parameter 25 is important to adapt the printer for the correct type of Centronics communication. Default setting is IBM/Epson type.
- Parameter 28 is intended for applications where high printout accuracy is required, e.g. when using very short labels. If the printhead is lifted, the paper will almost certainly be moved somewhat and the printout on the labels between the printhead and the LSS will not be positioned correctly. By chosing to clear the paper feed information when the printhead is lifted and then performing a **TESTFEED** to get new paper feed data, any such errors will be avoided.
- Parameter 32 is mainly used by service technicians.

For detailed explanations, please refer to the UBI Fingerprint 7.11 Reference Manual.

Example showing how the error type is set from the host and the new setting is read back:

```
10 INPUT "Error type: ", A%
20 SYSVAR(19)=A%                          (sets error type)
30 B%=SYSVAR(19)                          (reads error type)
40 PRINT "The error type is set to: "; B%
RUN
                                                  yields e.g.
Error type: 2
The error type is set to: 2
```

15.8 Printhead

In addition to the setup, four instructions can be used to check and control the thermal printhead.

> ☞ ***Setup Mode***
> Also see:
> • Chapter 15.6
> • Installation & Operation manual

SYSVAR

Two parameters in the system variable **SYSVAR** allows you to check the printhead, also see chapter 15.7:

`SYSVAR(20)`	returns if the printer is set up for direct thermal or transfer printing.
`SYSVAR(21)`	returns the printhead density in dots/mm.

HEAD

The **HEAD** function allows you to identify possible faulty dots by means of abnormal resistance values. This application is closely connected to the **SET FAULTY DOT** and **BARADJUST** statements, see below. Note that some printhead errors, e.g. cracked or dirty dots, will not be detected by this function, since only the resistance is measured.

SET FAULTY DOT

This statement is used to mark specified dots on the printhead as faulty, either manually or automatically in connection with a **HEAD** function. Then, using a **BARADJUST** statement (see below), you can adjust the location of picket fence bar codes so the dots marked as faulty will not affect the printing, i.e. the faulty dot(s) will be situated between the bars.

You can also revoke all previous **SET FAULTY DOT** statements by marking all dots as correct.

BARADJUST

This statement enables automatic horizontal relocation of picket fence bar codes within specified limits. The software will keep record of all dots marked as faulty (see **SET FAULTY DOT** above) and relocate the bar code as to place the spaces between the bars in line with the faulty dot(s). Thereby, it will be possible to use the printer pending printhead replacement.

Note that the **BARADJUST** statement cannot be used for ladder bar codes, stacked bar codes (e.g. Code 16K), bar codes with horizontal lines (e.g. DUN-14), EAN/UPC bar codes, or two-dimensional bar codes (e.g. PDF-417).

15.8 Printhead, cont'd.

This example shows how a program can be made that checks the printhead for faulty dots and warns the operator when a faulty dot is encountered. Pending printhead replacement, the bar code is repositioned to ensure continued readability. Such a program takes a few seconds to execute (there may be more than a thousand dots to check), so it is advisable either to restrict the dot check to the part of the printhead that corresponds to the location of the bar code, or to perform the test at startup only.

```
10    OPEN "console:" FOR OUTPUT AS 10
20    IF HEAD(-1)<>0 THEN GOTO 9000
30    BEEP:D1$="Printhead Error!":D2$="":GOSUB 2000
40    GOSUB 1000
50    BARADJUST 20,20
60    GOTO 9000
1000  FUNCTEST "HEAD",TMP$
1010  A$=":" : TMP%=INSTR(TMP$,A$)+1
1020  RETURN
1030  SET FAULTY DOT -1
1040  QMEAN%=HEAD(-7)
1050  QMIN%=QMEAN%*85\100
1060 QMAX%=QMEAN%*115\100
1070  FOR I%=0 TO WHEAD%-1
1080  QHEAD%=HEAD(I%)
1090  IF QHEAD%>QMAX% OR QHEAD%<QMIN% THEN SET FAULTY
      DOT I%
1100  NEXT
2000  PRINT #10 : PRINT #10, LEFT$(D1$,16)
2010  PRINT #10, LEFT$(D2$,16);
2020  RETURN
9000  PRPOS 200,20
9010  BARTYPE "CODE39"
9020  BARRATIO 2,1 : BARMAG 2
9030  BARHEIGHT 150
9040  PRBAR "1234567890"
9050  PRINTFEED
9060  END
```

15.9 Transfer Ribbon

SYSVAR

A number of parameters in the system variable **SYSVAR** can be used to check the transfer ribbon, also see chapter 15.7:

SYSVAR(13)	returns the value of the optional ribbon counter (some models only).
SYSVAR(20)	returns if the printer is set up for direct thermal or transfer printing.
SYSVAR(23)	returns if a transfer ribbon is fitted or not.

15.10 Memory Test

FUNCTEST

The FUNCTEST statement is used to perform the following tests and place the result in a string variable:

- Test of a memory card (DOS-formatted or non DOS-formatted).
- Test of the printhead in regard of number of dots, head lifted or possible errors.
- Test of the sysyems part of the printer's permanent memory (“Kernel”).
- Test of ROM SIMMs.

*Example using **FUNCTEST** on an EasyCoder 501 XP. The program takes a few seconds to execute:*

```
10    FUNCTEST "CARD", A$
20    FUNCTEST "HEAD", B$
30    FUNCTEST "KERNEL", C$
40    FUNCTEST "ROM1", D$
50    PRINT "CARDTEST:", A$
60    PRINT "HEADTEST:", B$
70    PRINT "KERNELTEST:", C$
80    PRINT "ROM1-TEST:", D$
RUN
```

yields e.g.:

```
CARDTEST: NO CARD
HEADTEST: HEAD OK,SIZE:1280 DOTS
KERNELTEST:          8E4791DC
ROM1-TEST:           NO ROM

Ok
```

FUNCTEST$

The **FUNCTEST$** function is very similar to the **FUNCTEST** statement and is used for the same purposes. Due to the different syntax, programming is more simple:

```
10    PRINT "CARDTEST:", FUNCTEST$ ("CARD")
20    PRINT "HEADTEST:", FUNCTEST$ ("HEAD")
30    PRINT "KERNELTEST:", FUNCTEST$ ("KERNEL")
40    PRINT "ROM1-TEST:", FUNCTEST$ ("ROM1")
RUN
```

yields e.g.:

```
CARDTEST: NO CARD
HEADTEST: HEAD OK,SIZE:1280 DOTS
KERNELTEST:          8E4791DC
ROM1-TEST:           NO ROM

Ok
```

15.11 Version Check

VERSION$

The **VERSION$** function returns one of three characteristics of the printer:

VERSION$(0)	returns the software version (e.g. "UBI Fingerprint 7.11")
VERSION$(0)	returns the printer family (e.g. "501XP").
VERSION$(0)	returns the CPU board generation (e.g. "hardware version 2.1").

This instruction allows you to create programs that will work with several different printer models. For example, you may use the **VERSION$** function to determine the type of printer and select the appropriate one of several different sets of setup parameters.

Example (sets the setup according to the type of printer):

```
10    A$=VERSION$(1)
20    IF A$="501" THEN GOTO 1000
30    IF A$="601" THEN GOTO 2000
40    IF A$="501XP" THEN GOTO 3000
50    IF A$="601XP" THEN GOTO 4000
60    .....
70    .....
1000  SETUP "SETUP501.SYS"
1010  GOTO 60
2000  SETUP "SETUP601.SYS"
2010  GOTO 60
3000  SETUP "SETUP501XP.SYS"
3010  GOTO 60
4000  SETUP "SETUP601XP.SYS"
4010  GOTO 60
```

16. Error-Handling

16.1 Standard Error-Handling

UBI Fingerprint is intended to be as flexible as possible. Thus, there are very few fixed error-handling facilities, but instead there are a number of tools for designing error-handling routines according to the demands of each application.

The following error-handling facilities are always available:

- ***Out-of-Media Detection***
  Provided the printhead is lowered, the firmware will check for three possible errors when either the <**Print**> or <**Feed**> key on the printer is pressed. If an error is detected, a message will appear in the display:
  - Error 1005 *(Out of paper)*
  - Error 1031 *(Next label not found)*
  - Error 1027 *(Out of ribbon – thermal transfer printers only)*

  After the error has been attended to, the error message can be cleared by pressing any of the keys.

- ***Syntax Check***
  Each program line or instruction that is received on the standard IN channel will be checked for possible syntax errors before it is accepted. Provided there is a working two-way communication[1], possible syntax errors will be transmitted to the host on the standard OUT channel, e.g. *"Feature not implemented"* or *"Font not found"*.

- ***Execution Check***
  Any program or hardware error that stops the execution will be reported on the standard OUT channel, provided there is a working two-way communication[1]. In case of program errors, the number of the line where the error occurred will also be reported, e.g. *"Field out of label in line 110"*. After the error has been corrected, the execution must be restarted by means of a new **RUN** statement, unless there is a routine for dealing with the error condition included in the program.

Error Messages

By means of the system variable **SYSVAR(19)**, see chapter 15.7, you can choose between four types of error messages as illustrated by the following examples using error #19:

1. *"Invalid font in line 10"* *(default)*
2. *"Error 19 in line 10: Invalid font"*
3. *"E19"*
4. *"Error 19 in line 10"*

[1]/. For a working two-way communication, three conditions must be fulfilled:
• Serial communication
• Std IN channel = Std OUT channel
• Verbosity enabled.

16.2. Tracing Programming Errors

TRON/TROFF
Large program can be difficult to grasp. If the program does not work as expected, it may depend on some programming error that prevents the program from being executed in the intended order. The **TRON** (Trace On) statement allows you to trace the execution. When the program is run, each line number will be returned on the standard OUT channel in the order of execution, provided you have a working two-way communication[1].

TROFF (Trace Off) disables **TRON**.

16.3 Creating an Error-Handling Routine

In most application programs, it is useful to include some kind of error-handler. Obviously, how comprehensive the error-handler needs to be depends on the application and how independent from the host the printer will work. In this chapter, we will explain the general principles and the related instructions and in chapter 16.4, you will find an example on how an error-handling program can be composed.

ON ERROR GOTO...
This statement is described in more detail in the chapter 5.8. It is used to branch the execution to a subroutine if any kind of error occurs when a program is run. The major benefit is that the program will not stop, but the error can be identified and dealt with. The execution can then be resumed at an appropriate program line.

ERR
The **ERR** function returns the reference number of an error that has occurred. The actual meaning of the numbers can be found in the chapter *"Error Messages"* in the UBI Fingerprint 7.11 Reference Manual.

ERL
The **ERL** function returns the number of the line on which an error has occurred.

RESUME
This statement is used resume the execution after the error has been taken care of in a subroutine. The execution can be resumed at the statement where the error occurred, at the statement immediately following the one where the error occurred, or at any other specified line. Also see chapter 5.8.

[1]/. For a working two-way communication, three conditions must be fulfilled:
• Serial communication
• Std IN channel = Std OUT channel
• Verbosity enabled.

3. Creating an Error-Handling Routine, cont'd.

Example:
The four instructions described above can be used to branch to a subroutine, identify the error, branch to a secondary subroutine where the error is cleared and resume the execution. In the example only one error condition 1019 "Invalid Font" is taken care of, but the same principles can be used for more errors. You can test the example by either adding a valid font name or lifting the printhead before running the program.

```
10    OPEN "console:" FOR OUTPUT AS 1
20    ON ERROR GOTO 1000
30    PRPOS 50,100
40    PRTXT "HELLO"
50    PRINTFEED
60    A%=TICKS+400
70    B%=TICKS
80    IF B%<A% THEN GOTO 70 ELSE GOTO 90
90    PRINT #1 : PRINT #1
100   END
1000  SOUND 880,50
1010  EFLAG%=ERR : ELINE%=ERL
1020  IF EFLAG%=1019 THEN GOTO 2000 ELSE GOTO 3000
2000  PRINT #1 : PRINT #1
2010  PRINT #1, "Font missing"
2020  PRINT #1, "in line ", ELINE%;
2030  FONT "SW030RSN" : MAG 2,2 : INVIMAGE
2040  RESUME
3000  PRINT #1 : PRINT #1
3010  PRINT #1, "Undefined error"
3020  PRINT #1, "Program Stops!";
3030  RESUME NEXT
```

☞ ***Logical Operators***
Also see:
• Chapter 4.9

PRSTAT

Another instruction that can be used in connection with error-handling is the **PRSTAT** function. In addition to returning the current position of the insertion point (see chapter 10.1), it can also return the printer's status in regard several conditions, using a logical operator:

`IF PRSTAT (AND 0)`	Ok
`IF PRSTAT (AND 1)`	Printhead lifted
`IF PRSTAT (AND 2)`	Label not removed (LTS only)
`IF PRSTAT (AND 4)`	Printer out of paper
`IF PRSTAT (AND 8)`	Printer out of transfer ribbon
`IF PRSTAT (AND 16)`	Printhead voltage too high
`IF PRSTAT (AND 32)`	Printer is feeding

Multiple simultaneous errors are indicated by the sum of the values for each error, e.g. if both the printhead is lifted (1) and the printer is out paper (4) and ribbon (8), it can be detected by:

```
IF PRSTAT (AND 13)
```

16.4 Error-Handling Program

ERRHAND.PRG Utility Program

The ERRHAND.PRG contains routines for handling errors, managing the keyboard and display, and for printing. Use ERRHAND.PRG to quickly get started with your programming.

By merging ERRHAND.PRG with your program, the latter can gain access to ERRHAND's subroutines. Do not use the lines 10–20 and 100000–1900200 in your program, since those line numbers are used by ERRHAND.PRG.

Example:

```
NEW
LOAD "XXX.PRG"
MERGE "ROM:ERRHAND.PRG"
RUN
```

If you have more than one application program that requires error-handling in your printer, you will save valuable memory space by keeping ERRHAND.PRG stored separately and merging it with the current program directly after loading, compared with merging ERRHAND.PRG with each program. The approximate size of ERRHAND.PRG is 4 kilobyte.

Variables and subroutines in ERRHAND.PRG that your program can use, or which you can modify, are:

Variables:

- **`NORDIS1$`** and **`NORDIS2$`** at line 10 contain the main display texts. You may replace them with your own text.
- **`DISP1$`** and **`DISP2$`** contain the actual text that will appear on the printer's display on line 1 and 2 respectively.

Subroutines:

- ***At line 160,000***

  The errors which normally may occur during printing are taken care of:

Error 1005	Out of paper
Error 1006	No field to print
Error 1022	Head lifted
Error 1027	Out of transfer ribbon
Error 1031	Next label not found

  The subroutine shows the last error that occurred, if any, and the line number where the error was detected. The information is directed to your terminal. Called by the statement **`GOSUB 160000`**.

- ***At line 200,000***

  Error-handling routines, which can be called from routines where error may occur, e.g.:

  ```
  IF EFLAG% < > 0 THEN GOSUB 200000
  ```

  The error-handling routine can be modified to handle other errors than those previously mentioned.

16.4 Error-Handling Program, cont'd.

ERRHAND.PRG Utility Program, cont'd.:

- ***At line 400,000***
  The **FEED**-routine executes a **FORMFEED** with error-checking. Called by the statement **GOSUB 400000**
- ***At line 500,000***
  The **PRINT**-routine executes a **PRINTFEED** with error-checking. Called by the statement **GOSUB 500000**
- ***At line 600,000***
  This subroutine clears the printer's display and makes the display texts stored in the variables **DISP1$** and **DISP2$** appear on the first and second line respectively in the display. Called by the statement **GOSUB 600000**
- ***At line 700,000***
  The Init routine initiates error-checking, opens the console for output and displays the main display texts (**NORDIS1$** and **NORDIS2$**). It also sets up the some of the keys on the keyboard (if any) and assigns subroutines to each key. Called by the statement **GOSUB 700000**
- ***At line 1,500,000***
  The <**Pause**> key (key No. 15) interrupts the program until the same key is pressed a second time. Called by the statement **GOSUB 1500000**
- ***At line 1,700,000***
  Routine for the <**Print**> key (key No. 17), that calls subroutine 500,000. Called by the statement **GOSUB 1700000**
- ***At line 1,800,000***
  Routine for the <**Setup**> key (key No. 18). Enters the Setup Mode of the printer. Called by the statement **GOSUB 1800000**
- ***At line 1,900,000***
  Routine for the <**Feed**> key (key No. 19), that calls subroutine 400,000. Called by the statement **GOSUB 1900000**

For more information, refer to the complete listing that follows.

16.4 Error-Handling Program, cont'd.

Listing of ERRHAND.PRG Utility Program

```
10      PROGNO$ = "Ver. 1.2 92-01-10"
15      NORDIS1$ = "TEST PROGRAM" : NORDIS2$ = "VERSION 1.2"
20      GOSUB 700000 : 'Initiate
100000  'Error routine
100010  EFLAG% = ERR
100050  'PRINT EFLAG%:'Activate for debug
100060  LASTERROR% = EFLAG%
100200  RESUME NEXT
160000  'PRINT "Last error = ";LASTERROR%: 'Activate for debug
160050  'IF LASTERROR% <> 0 THEN PRINT "At line ";ERL
160100  LASTERROR% = 0
160200  RETURN
200000  'Error handling routine
200010  IF EFLAG% = 1006 THEN GOTO 200040:'Formfeed instead of print
200020  LED (1) ON : LED (0) OFF : BUSY
200030  SOUND 400, 10
200040  IF EFLAG% = 1031 THEN GOSUB 300000
200050  IF EFLAG% = 1005 THEN GOSUB 310000
200060  IF EFLAG% = 1006 THEN GOSUB 320000
200070  IF EFLAG% = 1022 THEN GOSUB 330000
200080  IF EFLAG% = 1027 THEN GOSUB 340000
200090  DISP1$ = NORDIS1$ : DISP2$ = NORDIS2$
200100  GOSUB 600000
200110  LED (1) OFF : LED (0) ON : READY
200400  RETURN
300000  'Error 1031 Next label not found
300010  DISP1$ = "LABEL NOT FOUND"
300020  DISP2$ = "ERR NO. " + STR$ (ERR)
300030  GOSUB 600000
300040  EFLAG% = 0
300050  FORMFEED
300060  IF EFLAG% = 1031 THEN GOTO 300040
300200  RETURN
310000  'Error 1005 Out of paper
310010  DISP1$ = "OUT OF PAPER"
310020  DISP2$ = "ERR NO. " + STR$ (ERR)
310030  GOSUB 600000
310040  IF (PRSTAT AND 1)=0 THEN GOTO 310040:'Wait until head lifted
310050  EFLAG% = 0
310060  IF (PRSTAT AND 1) = 0 THEN FORMFEED ELSE GOTO 310060
310070  IF EFLAG% = 1005 THEN GOTO 310040
310080  IF EFLAG% = 1031 THEN GOSUB 300000
310200  RETURN
320000  'Error 1006 No field to print
320010  GOSUB 400000
320200  RETURN
```

16.4 Error-Handling Program, cont'd.

Listing of ERRHAND.PRG Utility Program, cont'd.

```
330000  'Error 1022 Head lifted
330010  DISP1$ = "HEAD LIFTED"
330020  DISP2$ = "ERR NO. " + STR$ (ERR)
330030  GOSUB 600000
330040  IF (PRSTAT AND 1) THEN GOTO 330040
330050  FORMFEED
330060  IF PCOMMAND% THEN GOSUB 500000
330200  RETURN
340000  'Error 1027 Out of transfer ribbon
340010  DISP1$ = "OUT OF RIBBON"
340020  DISP2$ = "ERR NO. " + STR$ (ERR)
340030  GOSUB 600000
340040  IF (PRSTAT AND 8) THEN GOTO 340040
340050  GOSUB 1500000
340200  IF PCOMMAND% THEN GOSUB 500000
349000  RETURN
400000  'Feed routine
400010  EFLAG% = 0
400020  FORMFEED
400200  IF EFLAG% <> 0 THEN GOSUB 200000
400300  RETURN
500000  'Print routine
500010  EFLAG% = 0
500020  PCOMMAND% = 1
500030  PRINTFEED
500040  IF EFLAG% <> 0 THEN GOSUB 200000
500100  PCOMMAND% = 0
500300  RETURN
600000  'Display handler
600010  PRINT # 10
600020  PRINT # 10
600030  PRINT # 10, DISP1$
600040  PRINT # 10, DISP2$;
600200  RETURN
700000  'Init routine
700010  ON ERROR GOTO 100000
700020  OPEN "console:" FOR OUTPUT AS # 10
700030  DISP1$ = NORDIS1$ : DISP2$ = NORDIS2$
700040  GOSUB 600000
700100  ON KEY (15) GOSUB 1500000 : 'PAUSE
700110  ON KEY (17) GOSUB 1700000 : 'PRINT
700120  ON KEY (18) GOSUB 1800000 : 'SETUP
700130  ON KEY (19) GOSUB 1900000 : 'FEED
700140  KEY (15) ON
700150  KEY (17) ON
700160  KEY (18) ON
700170  KEY (19) ON
700230  LED (0) ON
700240  LED (1) OFF
```

16.4 Error-Handling Program, cont'd.

Listing of ERRHAND.PRG Utility Program, cont'd.

```
700300  PAUSE% = 0
700500  RETURN
1500000 'Pause function
1500010 KEY (15) ON
1500020 PAUSE% = PAUSE% XOR 1
1500030 BUSY : LED (0) OFF
1500040 DISP1$ = "Press <PAUSE>" : DISP2$ = "to continue"
1500050 GOSUB 600000
1500060 IF PAUSE% = 0 THEN GOTO 1500100
1500070 SOUND 131, 2
1500080 SOUND 30000, 20
1500090 IF PAUSE% THEN GOTO 1500070
1500100 READY : LED (0) ON
1500110 DISP1$ = NORDIS1$ : DISP2$ = NORDIS2$
1500120 GOSUB 600000
1502000 RETURN
1700000 'Printkey
1700010 KEY (17) OFF
1700020 GOSUB 500000
1700030 KEY (17) ON
1700200 RETURN
1800000 'Setup key
1800010 KEY (18) OFF
1800020 LED (0) OFF
1800030 BUSY
1800040 SETUP
1800050 READY
1800060 LED (0) ON
1800080 KEY (18) ON
1800090 DISP1$ = NORDIS1$ : DISP2$ = NORDIS2$
1800100 GOSUB 600000
1800200 RETURN
1900000 'Feed key
1900010 KEY (19) OFF
1900020 GOSUB 400000
1900030 KEY (19) ON
1900200 RETURN
```

16.4 Error-Handling Program, cont'd.

Extensions to ERRHAND.PRG Utility Program

The following subroutines are not included in ERRHAND.PRG, but may be added manually to stop new input via the printer's keyboard while a subroutine is executed:

- Turn on all keys after having completed a subroutine by issuing the statement **GOSUB 800000**

```
800000 'Turn all keys on
800010 I% = 0
800020 IF I% > 21 THEN GOTO 800060
800030 KEY (I%) ON
800040 I% = I% + 1
800050 GOTO 800020
800060 RETURN
```

- Turn off all keys before entering a subroutine by issuing the statement **GOSUB 900000**

```
900000 'Turn all keys off
900010 I% = 0
900020 IF I% > 21 THEN GOTO 900060
900030 KEY (I%) OFF
900040 I% = I% + 1
900050 GOTO 900020
900060 RETURN
```

17. Reference Lists

17.1 Instructions in Alphabetical Order

Instruction	See chapter	Purpose
ABS	9.2	Returning the absolute value of a numeric expression.
ACTLEN	11.4	Returning the length of the most recently executed **PRINTFEED**, **FORMFEED** or **TESTFEED** statement.
ALIGN (AN)	10.1	Specifying which part (anchor point) of a text, bar code field, image field, line or box will be positioned at the insertion point.
ASC	9.2	Returning the decimal ASCII value of the first character in a string expression.
BARADJUST	15.8	Enabling/disabling automatic adjustment of bar code position in order to avoid faulty printhead dots.
BARFONT (BF)	10.3	Specifying fonts for the printing of bar code interpretation.
BARFONT (BF) ON/OFF	10.3	Enabling/disabling the printing of bar code interpretation.
BARHEIGHT (BH)	10.3	Specifying the height of a bar code.
BARMAG (BM)	10.3	Specifying the magnification in regard of width of the bars in a bar code.
BARRATIO (BR)	10.3	Specifying the ratio between the wide and the narrow bars in a bar code.
BARSET	10.3	Specifying a bar code and setting additional parameters to complex bar codes.
BARTYPE (BT)	10.3	Specifying the type of bar code.
BEEP	15.4	Ordering the printer to emit a beep.
BREAK	5.12	Specifying a break interrupt character separately for the keyboard and each serial communication channel.
BREAK ON/OFF	5.12	Enabling/disabling break interrupt separately for the keyboard and each serial communication channel.
BUSY	7.7	Ordering a busy signal, e.g. XOFF, CTS/RTS or PE, to be transmitted from the printer on the specified communication channel.
CHDIR	6.1	Specifying the current directory.
CHECKSUM	6.9	Calculating the checksum of a range of program lines in connection with the transfer of programs.
CHR$	9.2	Returning the readable character from a decimal ASCII code.
CLEANFEED	11.1	Running the printer's feed mechanism.
CLEAR	6.1	Clearing strings, variables and arrays to free memory space.
CLL	11.5	Partial or complete clearing of the print image buffer.
CLOSE	6.4, 7.3-7.6, 8.3-8.5	Closing one or several files and/or devices for input/output.
COM ERROR ON/OFF	7.8	Enabling/disabling error handling on the specified communication channel.
COMBUF$	7.8	Reading the data in the buffer of the specified communication channel.
COMSET	7.8	Setting the parameters for background reception of data to the buffer of a specified communication channel.
COMSET OFF	7.8	Turning off background data reception and emptying the buffer of the specified communication channel.
COMSET ON	7.8	Emptying the buffer and turning on background data reception on the specified communication channel.
COMSTAT	7.8	Reading the status of the buffer of the specified communication channel.
COPY	5.13, 6.2-6.4, 8.5	Copying files.
CSUM	6.10	Calculating the checksum of an array of strings.
CUT	11.3	Activating an optional paper cutting device.
CUT ON/OFF	11.3	Enabling/disabling automatic cutting after **PRINTFEED** execution and optionally adjusting the paper feed before and after the cutting.
DATE$	9.3, 15.5	Setting or returning the current date.
DATEADD$	9.3	Returning a new date after a number of days have been added to, or subtracted from, the current date or optionally a specified date.
DATEDIFF	9.3	Returning the difference between two dates as a number of days.
DELETE	5.4, 8.1	Deleting one or several consecutive program lines from the printer's working memory.

17.1 Instructions in Alphabetical Order, cont'd.

Instruction	See chapter	Purpose
DEVICES	4.10	Returning the names of all devices to the standard OUT channel.
DIM	6.10	Specifying the dimensions of an array.
DIR	10.1	Specifying the print direction.
END	5.4	Ending the execution of the current program or subroutine and closing all **OPENed** files and devices.
EOF	7.4	Checking for an end-of-file condition.
ERL	16.3	Returning the number of the line on which an error condition has occurred.
ERR	16.3	Returning the code number of an error that has occurred.
FIELD	7.5, 8.4	Creating a single-record buffer for a random file and dividing the buffer into fields to which string variables are assigned.
FIELDNO	11.5	Getting the current field number for partial clearing of the print buffer by a **CLL** statement.
FILE& LOAD	6.6, 12.2	Reception and storing of binary files in the printer's memory.
FILES	6.2, 8.1, 14.4	Listing the files stored in one of the printer's directories to the standard OUT channel.
FONT (FT)	10.2	Selecting a scalabele TrueType or TrueDoc single-byte font for the printing of the subsequent **PRTXT** statements.
FONTD	10.2	Selecting a scalabele TrueType or TrueDoc double-byte font for the printing of the subsequent **PRTXT** statements.
FONTNAME$	12.4	Returning the names of the fonts stored in the printer's memory.
FONTS	8.1, 12.4	Returning the names of all fonts stored in the printer's memory to the standard OUT channel.
FOR...TO...NEXT	5.9	Creating a loop in the program execution, where a counter is incremented or decremented until a specified value is reached.
FORMAT	6.1	Formatting the printer's permanent memory, or formatting a SRAM-type memory card to MS-DOS format.
FORMAT DATE$	9.3	Specifying the format of the string returned by **DATE$("F")** and **DATEADD$(...,"F")** instructions.
FORMAT TIME$	9.3	Specifying the format of the string returned by **TIME$("F")** and **TIMEADD$(...,"F")** instructions.
FORMFEED (FF)	11.1	Activating the paper feed mechanism in order to feed out or pull back a certain length of the paper web.
FRE	6.1	Returning the number of free bytes in the printer's temporary memory.
FUNCTEST	15.10	Performing various hardware tests.
FUNCTEST$	15.10	Performing various hardware tests.
GET	7.5	Reading a record from a random file to a random buffer.
GOSUB	5.7	Branching to a subroutine.
GOTO	5.6-5.7	Branching unconditionally to a specified line.
HEAD	15.8	Returning the result of a thermal printhead check.
IF..THEN...[ELSE]	5.5	Conditional execution controlled by the result of a numeric expression.
IMAGE LOAD	6.5, 14.3	Receiving, converting and installing image and font files.
IMAGENAME$	14.4	Returning the names of the images stored in the printer's memory.
IMAGES	8.1	Returning the names of all images stored in the printer's memory to the standard OUT channel.
IMMEDIATE ON/OFF	5.4	Enabling/disabling the immediate mode of UBI Fingerprint in connection with program editing without line numbers.
INKEY$	7.2	Reading the first character in the receive buffer of the standard IN channel.
INPUT (IP)	7.2	Receiving input data via the standard IN channel during the execution of a program.
INPUT#	7.3-7.6, 15.1	Reading a string of data from an **OPENed** device or sequential file.
INPUT$	7.2-7.6, 15.1	Returning a string of data, limited in regard of number of characters, from the standard IN channel, or optionally from an **OPENed** file or device.
INSTR	9.2	Searching a specified string for a certain character, or sequence of characters, and returning its position in relation to the start of the string.

17.1 Instructions in Alphabetical Order, cont'd.

Instruction	See chapter	Purpose
INVIMAGE (II)	10.2, 10.4	Inversing the printing of text and images from "black-on-white" to "white-on-black.
KEY BEEP	15.1	Resetting the frequency and duration of the sound produced by the beeper, when any key on the printer's keyboard is pressed down.
KEY ON/OFF	15.1	Enabling/disabling a specified key on the printer's front panel to be used in connection with an **ON KEY...GOSUB** statement.
KEYBMAP$	15.1	Returning or setting the keyboard map table.
KILL	5.13, 6.3-6.4	Deleting a file from the printer's memory or from a DOS-formatted SRAM memory card inserted in the memory card adapter.
LAYOUT	10.7	Handling of layout files.
LBLCOND	11.1	Overriding the paper feed setup.
LED ON/OFF	15.3	Turning a specified LED control lamp on or off.
LEFT$	9.2	Returning a specified number of characters from a given string starting from the extreme left side of the string, i.e. from the start.
LEN	9.2	Returning the number of character positions in a string.
LET	4.7	Assigning the value of an expression to a variable.
LINE INPUT	7.2	Assigning an entire line, including punctuation marks, from the standard IN channel to a single string variable.
LINE INPUT#	7.3-7.6, 15.1	Assigning an entire line, including punctuation marks, from a sequential file or a device to a single string variable.
LIST	5.4, 6.3 8.1	Listing the current program completely or partially, or listing all variables, to the standard OUT channel.
LOAD	5.13, 6.3	Loading a copy of a program, residing in the current directory or in another specified directory, into the printer's working memory.
LOC	6.4, 7.4-7.5, 7.8, 8.3-8.5	Returning the current position in an **OPENed** file or the status of the buffers in an **OPENed** communication channel.
LOF	6.4, 7.4-7.5, 7.8, 8.3-8.5	Returning the length in bytes of an **OPENed** sequential or random file or returning the status of the buffers in an **OPENed** communication channel.
LSET	8.4	Placing data left-justified into a field in a random file buffer.
LTS& ON/OFF	11.3	Enabling or disabling the label taken sensor.
MAG	10.2, 10.4	Magnifying a font, barfont or image up to four times separately in regard of height and width.
MAP	9.1	Changing the ASCII value of a character when received on the standard IN channel, or optionally on another specified communication channel.
MERGE	6.3	Merging a program in the printer's current directory, or optionally in another specified directory, with the program currently residing in the printer's working memory.
MID$	9.2	Returning a specified part of a string.
NAME DATE$	9.3	Formatting the month parameter in return strings of **DATE$("F")** and **DATEADD$(...,"F")**.
NAME WEEKDAY$	9.3	Formatting the day parameter in return strings of **WEEKDAY$**.
NASC	9.1	Selecting a single-byte character set.
NASCD	9.1	Selecting a double-byte character set according to the Unicode standard.
NEW	5.4, 6.3	Clearing the printer's working memory in order to allow a new program to be created.
NORIMAGE (NI)	10.2, 10.5	Returning to normal printing after an **INVIMAGE** statement has been issued.
ON BREAK GOSUB	5.8, 5.12	Branching to a subroutine, when a break interrupt instruction is received.
ON COMSET GOSUB	5.8, 7.8	Branching to a subroutine, when the background reception of data on the specified communication channel is interrupted.
ON ERROR GOTO	5.8, 16.3	Branching to an error-handling subroutine when an error occurs.
ON GOSUB	5.8	Conditional branching to one or several subroutines.
ON GOTO	5.8	Conditional branching to one of several lines.
ON KEY GOSUB	5.8, 15.1	Branching to a subroutine when a specified key on the printer's front panel is activated.

17.1 Instructions in Alphabetical Order, cont'd.

Instruction	See chapter	Purpose
ON/OFF LINE	7.7	Controlling the SELECT signal on the Centronics communication channel.
OPEN	6.4, 7.3-7.6, 8.3-8.5, 15.2	Opening a file or device – or creating a new file – for input, output or append, allocating a buffer and specifying the mode of access.
OPTIMIZE "BATCH" ON/OFF	11.5	Enabling/disabling optimizing for batch printing.
PCX2BMP	6.5, 14.3	Converting and intalling image files in .PCX format.
PORTIN	7.10	Reading the status of a port on the Industrial Interface Board.
PORTOUT ON/OFF	7.10	Setting one of four relay port or one of eight optical ports on an Industrial Interface Board to either on or off.
PRBAR (PB)	10.3	Providing input data to a bar code.
PRBOX (PX)	10.5	Creating a box.
PRIMAGE (PM)	10.4	Selecting an image stored in the printer's memory.
PRINT (?)	8.1	Printing data to the standard OUT channel.
PRINT KEY ON/OFF	11.3	Enabling/disabling printing of a label by pressing the Print key.
PRINT#	8.3, 8.5, 15.2	Printing of data to a specified **OPENed** device or sequential file.
PRINTFEED (PF)	11.3	Printing and feeding out one or a specified number of labels, tickets, tags or portions of strip, according to the printer's setup.
PRINTONE	8.1	Printing characters specified by their ASCII values to the standard OUT channel.
PRINTONE#	8.3, 8.5	Printing characters specified by their ASCII values to a device or sequential file.
PRLINE (PL)	10.6	Creating a line.
PRPOS (PP)	10.1	Specifying the insertion point for a line of text, a bar code, an image, a box, or a line.
PRSTAT	10.1, 16.3	Returning the printer's current status or, optionally, the current position of the insertion point.
PRTXT (PT)	10.2	Providing the input data for a text field, i.e. a line of text.
PUT	8.4	Writing a given record from the random buffer to a given random file.
RANDOM	9.4	Generating a random integer within a specified interval.
RANDOMIZE	9.4	Reseeding the random number generator, optionally with a specified value.
READY	7.7	Ordering ready signal, e.g. XON, CTS/RTS or PE, to be transmitted from the printer on the specified communication channel.
REBOOT	5.14	Restarting the printer.
REDIRECT OUT	6.4, 8.2	Redirecting the output data to a created file.
REM (')		Adding headlines and explanations to a program without including them in the execution.
REMOVE IMAGE	12.2-12.3, 14.4	Removing a specified image from the printer's memory.
RENUM	5.4	Renumbering the lines of the program currently residing in the printer's working memory.
RESUME	5.8, 16.3	Resuming program execution after an error-handling subroutine has been executed.
RETURN	5.7	Returning to the main program after having branched to a subroutine because of a **GOSUB** statement.
RIGHT$	9.2	Returning a specified number of characters from a given string starting from the extreme right side of the string, i.e. from the end.
RSET	8.4	Placing data right-justified into a field in a random file buffer.
RUN	5.11, 6.3	Starting the execution of a program.
SAVE	5.13, 6.3	Saving a file in the printer's memory or optionally in a DOS-formatted memory card.
SET FAULTY DOT	15.8	Marking one or several dots on the printhead as faulty, or marking all faulty dots as correct.
SETSTDIO	7.1	Selecting standard IN and OUT communication channel.
SETUP	15.6	Entering the printer's Setup Mode, changing the setup by means of a setup file or setup string, or creating a setup file containing the printer's current setup values.
SGN	9.2	Returning the sign (positive, zero or negative) of a specified numeric expression.

17.1 Instructions in Alphabetical Order, cont'd.

Instruction	See chapter	Purpose
SORT	6.10	Sorting a one-dimensional array.
SOUND	15.4	Making the printer's beeper produce a sound specified in regard of frequency and duration.
SPACE$	9.2	Returning a specified number of space characters.
SPLIT	6.10	Splitting a string into an array according to the position of a specified separator character and returning the number of elements in the array.
STORE IMAGE	14.3	Setting up parameters for storing an image in the printer's memory.
STORE INPUT	14.3	Receiving and storing protocol frames of image data in the printer's memory.
STORE OFF	14.3	Terminating the storing of an image and resetting the storing parameters.
STR$	9.2	Returning the string representation of a numeric expression.
STRING$	9.2	Repeatedly returning the character of a specified ASCII value, or the first character in a specified string
SYSVAR	7.7, 14.3, 15.7-15.9, 16.1	Reading or setting various system variables.
TESTFEED	11.1	Adjusting the label stop sensor while perfoming a number of formfeeds.
TICKS	9.3	Returning the time that has passed since the last power-up in the printer, expressed in number of “Ticks” (1 Tick = 0.01 seconds).
TIME$	9.3, 15.5	Setting or returning the current time.
TIMEADD$	9.3	Returning a new time after a number of seconds have been added to, or subtracted from, the current time or optionally a specified time.
TIMEDIFF	9.3	Returning the difference in number of seconds between two specified moments of time in number of seconds.
TRANSFER KERMIT	6.8	Transferring of data files using Kermit communication protocol.
TRANSFER STATUS	6.8	Checking last **TRANSFER KERMIT** operation.
TRANSFER$	6.4	Executing a transfer from source to destination as specified by a **TRANSFERSET** statement.
TRANSFERSET	6.4	Entering setup for the **TRANSFER$** function.
TRON/TROFF	16.2	Enabling/disabling tracing of the program execution.
VAL	9.2	Returning the numeric representation of a string expression.
VERBON/VERBOFF	7.7	Specifying the verbosity level of the communication from the printer on the standard OUT channel (serial communication only).
VERSION$	15.11	Returning the version of the firmware, printer family, or type of CPU board
WEEKDAY	9.3	Returning the weekday of a specified date.
WEEKDAY$	9.3	Returning the name of the weekday from a specified date.
WEEKNUMBER	9.3	Returning the number of the week for a specified date.
WHILE...WEND	5.9	Executing a series of statements in a loop providing a given condition is true.

17.2 Instructions by Field of Application

Instruction	Abbr.	Type	Purpose
SETUP AND PREFERENCES			
General UBI Fingerprint Control:			
CHDIR<scon>		Stmt	Change current directory
MAP[<nexp>,]<nexp>,<nexp>		Stmt	Remapping
NASC<nexp>		Stmt	Select single-byte character set
NASCD<nexp>		Stmt	Select double-byte character set
REBOOT		Stmt	Restart printer
SETUP [[WRITE<sexp>] | [<sexp>]| [<sexp>]]		Stmt	Printer setup
SYSVAR(<nexp>)		Array	Read or set various system variables
Setting the Clock/Calendar:			
DATE$=<sexp>		Var	Set the date
TIME$=<sexp>		Var	Set the time
OPERATOR INTERFACE			
Keyboard Setup:			
KEY(<nexp>)ON|OFF		Stmt	Enable/disable key on printer's keyboard
ON KEY(<nexp>)GOSUB<ncon>|<line label>		Stmt	Key-initiated branching
KEY BEEP<nexp>,<nexp>		Stmt	Set frequency and duration of key response
KEYBMAP$(<nexp>)=<sexp>		Var	Set the keyboard map table
Output to Display:			
OPEN "console:" FOR OUTPUT AS[#]<nexp>		Stmt	Open display for output
PRINT#<nexp>[,<<nexp>|<sexp>>[<,|;><<nexp|<sexp>>...][;]]		Stmt	Print data to display
CLOSE [#]<nexp>		Stmt	Close display for output
LED Control Lamps:			
LED<nexp>ON|OFF		Stmt	Turn LED on or off
Audible Signals:			
BEEP		Stmt	Emit a beep
SOUND<nexp>,<nexp>		Stmt	Produce sound
Breaking Program Execution:			
BREAK<nexp>,<nexp>		Stmt	Specify break interrupt character
BREAK <nexp> ON|OFF		Stmt	Enable/disable break interrupt
ON BREAK<nexp>GOSUB<ncon>|<line label>		Stmt	Branching at break interrupt
PRINTER CHECKOUT AND CONTROL			
Keyboard:			
<svar> = KEYBMAP$(<nexp>)		Var	Read keyboard mapping
Memory:			
CLEAR		Stmt	Clear strings, variables and arrays
FORMAT<sexp>[,<nexp>[,<nexp>]][,A]		Stmt	Format "c:" memory or "card1:"
FRE(<<nexp>|<sexp>>)		Func	Return number of free bytes in "tmp:"
FUNCTEST<sexp>,<svar>		Stmt	Testing the hardware
FUNCTEST$(<sexp>)		Func	Testing the hardware
KILL<sexp>		Stmt	Delete file
REMOVE IMAGE<sexp>		Stmt	Remove image from memory
Odometer:			
SYSVAR(32)		Array	Read kilometre counter
Printhead:			
BARADJUST<nexp>,<nexp>		Stmt	Enable/disable auto bar code repositioning
HEAD(<nexp>)		Func	Checking printhead dots
FUNCTEST<sexp>,<svar>		Stmt	Checking printhead
FUNCTEST$(<sexp>)		Func	Checking the printhead
SET FAULTY DOT<nexp>[,<nexp>...]		Stmt	Marking dots as faulty for BARADJUST
SYSVAR(21|22)		Array	Read printhead density or number of dots
Transfer Ribbon:			
SYSVAR(13|20|23)		Array	Read counter, mode or ribbon end sensor

17.2 Instructions by Field of Application, cont'd.

Instruction	Abbr.	Type	Purpose
PROGRAMMING:			
Managing Programs and Files:			
CHECKSUM(<nexp>,<nexp>)		Func	Calculate checksum at program transfer
COPY<sexp>[,<sexp>]		Stmt	Copy file
KILL<sexp>		Stmt	Delete file
LOAD<scon>		Stmt	Load program
MERGE<scon>		Stmt	Merge programs
NEW		Stm	Clear the working memory
SAVE<scon>[,P|L]		Stmt	Save program
Listings:			
DEVICES		Stmt	List devices to standard I/O channel
FILES[<sexp>][,A]		Stmt	List files to standard I/O channel
FONTNAME$(<nexp>)		Func	Return names of fonts in printer's memory
FONTS		Stmt	List all fontnames to standard I/O channel
IMAGENAME$(<nexp>)		Func	Return names of images in printer's memory
IMAGES		Stmt	List all imagenames to standard I/O channel
LIST[[<ncon>[- <ncon>]]|,V]		Stmt	List current program or all variables to std I/O
VERSION$[(<nexp>)]		Func	Returns F/W or H/W version or printer model
Program Editing and Execution:			
DELETE<ncon>[-<ncon>]		Stmt	Delete program lines
END		Stmt	Terminate program execution
IMMEDIATE ON|OFF		Stmt	Start/stop writing program w/o line numbers
LIST[[<ncon>[- <ncon>]]|,V]		Stmt	List current program or all variables to std I/O
NEW		Stmt	Clear the working memory
REM<remark>	'	Stmt	Remark
RENUM[<ncon>][,[<ncon>][,<ncon>]]		Stmt	Renumber program lines
RUN[<<scon>|<ncon>>]		Stmt	Execute program
SAVE<scon>[,P|L]		Stmt	Save program
Data Manipulation:			
ABS(<nexp>)		Func	Return the absolute value of an expression
ASC(<sexp>)		Func	Return ASCII code for 1:st char. in string
CHR$(<nexp>)		Func	Convert ASCII code
INSTR([<nexp>,]<sexp>,<sexp>]		Func	Return position of character in string
LEFT$(<sexp>,<nexp>)		Func	Return characters from left side of string
LEN(<sexp>)		Func	Return number of characters in string
[LET]<<nvar>=<nexp>>|<<svar>=<sexp>>		Stmt	Assign a value to a variable
MID$(<sexp>,<nexp>[,<nexp>])		Func	Return part of string
RANDOM (<nexp>,<nexp>)		Func	Generate a random integer
RANDOMIZE[<nexp>]		Stmt	Reseed random number generator
RIGHT$(<sexp>,<nexp>)		Func	Return characters from right side of string
SGN(<nexp>)		Func	Return sign of numeric expression
SPACE$(<nexp>)		Func	Return specified number of space characters
STR$(<nexp>)		Func	Return string representation of num. expr.
STRING$(<nexp>,<<nexp>|<sexp>>)		Func	Return a number of repeated characters
VAL(<sexp>)		Func	Return numeric representation of string expr.
Branching and Conditionals:			
FOR<nvar>=<nexp>TO<nexp>[STEP<nexp>)]NEXT[<nvar>]		Stmt	Creating a program loop
GOSUB<ncon>|<line label>		Stmt	Branch to subroutine
GOTO<ncon>|<line label>		Stmt	Unconditional branching
IF<nexp>[,]THEN<stmt>[ELSE<stmt>]		Stmt	Conditional execution
ON <nexp>GOSUB<ncon>|<line label>[,<ncon>|<line label>...]		Stmt	Cond. branching to one of many subroutines
ON <nexp>GOTO<ncon>|<line label>[,<ncon>|<line label>...]		Stmt	Conditional branching to one of several lines
RETURN[<ncon>|<line label>]		Stmt	Return from subroutine
WHILE<nexp>↵ <stmt>↵ [...<stmt>]↵ WEND		Stmt	Conditional execution of loop of statements

17.2 Instructions by Field of Application, cont'd.

Instruction	Abbr.	Type	Purpose
PROGRAMMING, cont'd:			
Arrays:			
CSUM<ncon>,<svar>,<nvar>		Stmt	Calculate checksum of array of strings
DIM<<nvar>|<svar>>(<nexp>[,<nexp>...])...[,<<nvar>|<svar>>(nexp>[,<nexp>...])]		Stmt	Set array dimensions
SORT<<nvar>|<svar>>,<nexp>,<nexp>,<nexp>		Stmt	Sort a one-dimensional array
SPLIT(<sexp>,<sexp>,<nexp>)		Func	Split a string into an array
Clock/Calendar Facilities:			
<svar>=DATE$[("F")]		Var	Read the date
<svar>=TIME$[("F")]		Var	Read the time
DATEADD$[(<sexp>,]<nexp>[,"F"])		Func	Add days to a date
TIMEADD$[(<sexp>,]<nexp>[,"F"])		Func	Add seconds to a time
DATEDIFF(<sexp>,<sexp>)		Func	Calculate difference between dates
TIMEDIFF(<sexp>,<sexp>)		Func	Calculate difference between times
FORMAT DATE$<sexp>		Stmt	Specify date format
FORMAT TIME$<sexp>		Stmt	Specify time format
NAME DATE$<nexp>,<sexp>		Stmt	Specify names of the months
NAME WEEKDAY$<nexp>,<sexp>		Stmt	Specify names of the weekdays
WEEKDAY(<sexp>)		Func	Return weekday of a date
WEEKDAY$(<sexp>)		Func	Return name of the weekday for a date
WEEKNUMBER(<sexp>)		Func	Return weeknumber for a date
TICKS		Func	Return time passed since startup
Error-handling:			
ERL		Func	Return number of line with error
ERR		Func	Return error code number
ON ERROR GOTO<ncon>|<line label>		Stmt	Branch at error
PRSTAT[(<nexp>)]		Func	Returns printer status or current X/Y position
RESUME[<<ncon>|<line label>|<NEXT>|<0>>]		Stmt	Resume program execution after error
SYSVAR(19)		Array	Set or return type of error message
TRON		Stmt	Enable tracing of program execution
TROFF		Stmt	Disable tracing of program execution
COMMUNICATION:			
Communication Control:			
BUSY[<nexp>]		Stmt	Send busy signal on communication channel
OFF LINE<nexp>		Stmt	SELECT signal low (Centronics)
ON LINE<nexp>		Stmt	SELECT signal high (Centronics)
READY[<nexp>]		Stmt	Send ready signal on communication channel
REDIRECT OUT[<sexp>]		Stmt	Redirect output data to file
SETSTDIO<nexp>[,<nexp>]		Stmt	Set standard I/O channels
SYSVAR(18)		Array	Set verbosity level
SYSVAR(25)		Array	Select type of Centronics communication
VERBOFF		Stmt	Verbosity off
VERBON		Stmt	Verbosity on
Background Communication:			
COM ERROR<nexp>ON|OFF		Stmt	Enable/disable error handling
COMBUF$(<nexp>)		Func	Read communication buffer
COMSET<nexp>,<sexp>,<sexp>,<sexp>,<sexp>,<nexp>		Stmt	Set communication parameters
COMSET<nexp>ON|OFF		Stmt	Turn on/off background data reception
COMSTAT(<nexp>)		Func	Read communication buffer status
ON COMSET<nexp>GOSUB<nexp>|<line label>		Stmt	Branch at background comm. interrupt

17.2 Instructions by Field of Application, cont'd.

Instruction	Abbr.	Type	Purpose
FILE TRANSFER:			
Binary Files:			
FILE& LOAD[<nexp>,]<sexp>,<nexp>[,<nexp>]		Stmt	Receive and store binary files
TRANSFER K[ERMIT]<sexp>[,<sexp>[,<sexp>[,<sexp>]]]		Stmt	Data transfer using Kermit protocol
TRANSFER S[TATUS]<nvar>,<svar>		Stmt	Check last TRANSFER KERMIT execution
Data Files:			
TRANSFER$(<nexp>)		Func	Execute transfer and set time-out
TRANSFERSET[#]<nexp>,[#]<nexp>,<sexp>[,<nexp>]		Stmt	Enter setup for file transfer using TRANSFER$
Font Files:			
FILE& LOAD[<nexp>,]<sexp>,<nexp>[,<nexp>]		Stmt	Receive and store font files (installed after restart)
IMAGE LOAD[<nexp>,]<sexp>,<nexp>,<sexp>[,<nexp>]		Stmt	Receive, convert and install fonts
TRANSFER K[ERMIT]<sexp>[,<sexp>[,<sexp>[,<sexp>]]]		Stmt	Transfer, convert and install fonts
TRANSFER S[TATUS]<nvar>,<svar>		Stmt	Check last TRANSFER KERMIT execution
Image Files:			
IMAGE LOAD[<nexp>,]<sexp>,<nexp>,<sexp>[,<nexp>]		Stmt	Receive, convert and install .PCX images
RUN "pcx2bmp [-i] [-v] <scon>[<scon>]"		–	Convert and install image files in .PCX format
STORE IMAGE[RLL][KILL]<sexp>,<nexp>,<nexp>,[<nexp>],<sexp>		Stmt	Set up image storage parameters
STORE INPUT<nexp>[,<nexp>]		Stmt	Receiving and storing image data
STORE OFF		Stmt	End storing of image data
SYSVAR(16|17)		Array	Read no. of bytes/frames received
INPUT TO UBI FINGERPRINT			
Input from Standard IN Channel:			
INKEY$		Func	Read 1:st character from std IN channel
INPUT[<scon><;|,>]<<nvar>|<svar>>[,<<nvar>|<svar>>...]	IP	Stmt	Input to variables
INPUT$(<nexp>[,<nexp>])		Func	Input, limited no. of characters
LINE INPUT[<scon>;]<svar>		Stmt	Input, entire line
Input from Host on Any Channel:			
CLOSE[[#]<nexp>[,[#]<nexp>...]]		Stmt	Close device
INPUT#<nexp>,<<nvar>|<svar>>[,<<nvar>|<svar>...]		Stmt	Input to variables
INPUT$(<nexp>[,<nexp>])		Func	Input, limited no. of characters
LINE INPUT#<nexp>,<svar>		Stmt	Input, entire line
LOC(<nexp>)		Func	Remaining no. of characters in receive buffer
LOF(<nexp>)		Func	Remaining free space in receive buffer
OPEN<sexp>FOR INPUT AS[#]<nexp>		Stmt	Open device
Input from Sequential File:			
CLOSE[[#]<nexp>[,[#]<nexp>...]]		Stmt	Close file
EOF(<nexp>)		Func	End of file
INPUT#<nexp>,<<nvar>|<svar>>[,<<nvar>|<svar>...]		Stmt	Input to variables
INPUT$(<nexp>[,<nexp>])		Func	Input, limited no. of characters
LINE INPUT#<nexp>,<svar>		Stmt	Input, entire line
LOC(<nexp>)		Func	Return current position in file
LOF(<nexp>)		Func	Return length of file
OPEN<sexp>FOR INPUT AS[#]<nexp>		Stmt	Open file
Input from Random File:			
CLOSE[[#]<nexp>[,[#]<nexp>...]]		Stmt	Close file
FIELD[#]<nexp>,<nexp>AS<svar>[,<nexp>AS<svar>...]		Stmt	Create a buffer for a random file
GET[#]<nexp>,<nexp>		Stmt	Read rec. from random file to random buffer
LOC(<nexp>)		Func	Return current position in file or buffer
LOF(<nexp>)		Func	Return length of file
OPEN<sexp>AS[#]<nexp>[LEN=<nexp>]		Stmt	Open a random file
Input from Printer's Keyboard:			
CLOSE [#]<nexp>		Stmt	Close keyboard for input
INPUT#<nexp>,<<nvar>|<svar>>[,<<nvar>|<svar>...]		Stmt	Input to variables
INPUT$(<nexp>[,<nexp>])		Func	Input, limited no. of characters
LINE INPUT#<nexp>,<svar>		Stmt	Input , entire line
OPEN"console:" FOR INPUT AS[#]<nexp>		Stmt	Open keyboard for input

17.2 Instructions by Field of Application, cont'd.

Instruction	Abbr.	Type	Purpose
INPUT TO UBI FINGERPRINT, cont'd:			
Industrial Interface:			
PORTIN(<nexp>)		Func	Reading status of a specified port
PORTOUT(<nexp>)ON|OFF		Stmt	Set the relay on a specified port
OUTPUT FROM UBI FINGERPRINT			
Output to Standard OUT Channel :			
PRINT[<<nexp>|<sexp>>[<,|;><<nexp>|<sexp>>...][;]]	?	Stmt	Print data to standard I/O channel
PRINTONE[<nexp>[<,|;><nexp>...][;]]		Stmt	Print ASCII characters to std I/O channel
Output to Any Communication Channel:			
CLOSE[[#]<nexp>[,[#]<nexp>...]]		Stmt	Close device
PRINT#<nexp>[,<<nexp>|<sexp>>[<,|;><<nexp|<sexp>>...][;]]		Stmt	Print data to device
PRINTONE#<nexp>[,<nexp>[<,|;><nexp>...][;]]		Stmt	Print ASCII characters to device
LOC(<nexp>)		Func	Remaining free bytes in transmitter buffer
LOF(<nexp>)		Func	Remaining no. of char. in transmitter buffer
OPEN<sexp>[FOR <OUTPUT|APPEND>]AS[#]<nexp>		Stmt	Open device
Output to a Sequential File:			
CLOSE[[#]<nexp>[,[#]<nexp>...]]		Stmt	Close file
PRINT#<nexp>[,<<nexp>|<sexp>>[<,|;><<nexp|<sexp>>...][;]]		Stmt	Print data to sequential file
PRINTONE#<nexp>[,<nexp>[<,|;><nexp>...][;]]		Stmt	Print ASCII characters to sequential file
LOC(<nexp>)		Func	Current position in file
LOF(<nexp>)		Func	Length of file
OPEN<sexp>[FOR <OUTPUT|APPEND>]AS[#]<nexp>		Stmt	Open file
Output to Random File:			
CLOSE[[#]<nexp>[,[#]<nexp>...]]		Stmt	Close file
FIELD[#]<nexp>,<nexp>AS<svar>[,<nexp>AS<svar>...]		Stmt	Create a buffer for a random file
LOC(<nexp>)		Func	Current position in file
LOF(<nexp>)		Func	Length of file
LSET<svar>=<sexp>		Stmt	Place data in random file buffer (left justified)
PUT[#]<nexp>,<nexp>		Stmt	Write rec. from random buffer to random file
OPEN<sexp>AS[#]<nexp>[LEN=<nexp>]		Stmt	Open a random file
RSET<svar>=<sexp>		Stmt	Place data in random file buffer (right justified)
FORMATTING AND PRINTING			
General Formatting Instructions:			
ALIGN<nexp>	AN	Stmt	Alignment
DIR<nexp>		Stmt	Select print direction
PRPOS<nexp>,<nexp>	PP	Stmt	Set coordinates for insertion point
LAYOUT[F,]<sexp>,<sexp>,<svar>|<sexp>,<nvar>|<sexp>		Stmt	Creating and using layout files
Text Printing:			
INVIMAGE	II	Stmt	Inverse image printing
MAG<nexp>,<nexp>		Stmt	Magnification of font (obsolete)
NORIMAGE	NI	Stmt	Return to normal image printing
FONT<sexp>[,<nexp>[,<nexp>]]	FT	Stmt	Select single-byte font
FONTD<sexp>[,<nexp>[,<nexp>]]		Stmt	Select double-byte font
PRTXT<<nexp>|<sexp>>[;<<nexp>|<sexp>>...][;]	PT	Stmt	Input data to text field
Bar Code Printing:			
BARFONT[#<ncon>,]<sexp>[,nexp>[,<nexp>[,<nexp>[,nexp>[,nexp]]]]][ON]	BF	Stmt	Specify bar code interpretation fonts
BARFONT ON	BF ON	Stmt	Enable bar code Interpretation
BARFONT OFF	BF OFF	Stmt	Disable bar code interpretation
BARHEIGHT<nexp>	BH	Stmt	Bar code height
BARMAG<nexp>	BM	Stmt	Bar code magnification
BARRATIO<nexp>,<nexp>	BR	Stmt	Wide/narrow bar ratio
BARSET[#<ncon>,][<sexp>[,<nexp>[,<nexp>[,<nexp>[,<nexp>[,<nexp>[,<nexp>[,<nexp>[,<nexp>[,<nexp>]]]]]]]]]		Stmt	Specifying complex bar codes
BARTYPE<sexp>	BT	Stmt	Bar code type
MAG<nexp>,<nexp>		Stmt	Magnification of barfont (obsolete)
PRBAR<<sexp>|<nexp>>	PB	Stmt	Input data to bar code field

17.2 Instructions by Field of Application, cont'd.

Instruction	Abbr.	Type	Purpose
FORMATTING AND PRINTING, cont'd:			
Image and Graphics Printing:			
INVIMAGE	II	Stmt	Inverse image printing
MAG<nexp>,<nexp>		Stmt	Magnification of image
NORIMAGE	NI	Stmt	Return to normal image printing
PRBOX<nexp>,<nexp>,<nexp>	PX	Stmt	Create a box
PRIMAGE<sexp>	PM	Stmt	Select a preprogrammed image
PRLINE<nexp>,<nexp>	PL	Stmt	Create a line
Printing and Paper Feed Control:			
ACTLEN		Func	Read length of last paper feed
CLEANFEED<nexp>		Stmt	Running the printer's feed mechanism
CLL[<nexp>]		Stmt	Clear print buffer
CUT		Stmt	Activate optional cutting device
CUT <nexp> ON|OFF		Stmt	Enable/disable automatic cut-off
FIELDNO		Func	Get current field number for CLL
FORMFEED[<nexp>]	FF	Stmt	Paper feed
LBLCOND<nexp>,<nexp>		Stmt	Overriding paper feed setup
LTS& ON|OFF		Stmt	Enable/disable label taken sensor
OPTIMIZE "BATCH" ON|OFF		Stmt	Enable/disable optimizing for batch printing
PRINT KEY ON|OFF		Stmt	Enable/disable PRINTFEED using Print key
PRINTFEED<nexp>	PF	Stmt	Print and feed out label or batch of labels
SYSVAR(28)		Array	Erase paper feed data
TESTFEED		Stmt	Auto adjustment of label stop sensor