UnForm® User Guide

Version 8.0

UnForm is published under license by:
Synergetic Data Systems, Inc.
3976 Durock Road, Suite 102
Shingle Springs, CA 95682
USA

Phone: (530)-672-9970
Fax: (530)-672-9975
Email: sdsi@synergetic-data.com
Web page: http://synergetic-data.com

UnForm® Document Management Software
License Agreement

NOTICE: OPENING THIS PACKAGE INDICATES YOUR ACCEPTANCE OF THE FOLLOWING TERMS AND CONDITIONS. PLEASE READ THEM. IF YOU DO NOT AGREE WITH THEM, RETURN THE PACKAGE UNOPENED, AND RETURN OR DESTROY ANY COPIES OF THE PROGRAM IN YOUR POSSESSION. THE DEALER FROM WHOM YOU PURCHASED THE SOFTWARE WILL REFUND YOUR PURCHASE PRICE.

"Program", as used herein, refers to both this documentation and the software programs described by this documentation.
"Developer", as used herein, refers to Allen D. Miglore. "Publisher" as used herein refers to Synergetic Data Systems, Inc.

LICENSE
You may use the Program on a single machine, and you may copy the Program into any machine-readable format for backup purposes only. If you transfer the Program to another machine, you agree to destroy the Program, together with all copies, in whole or in part, on the original machine.

You may not copy, modify, or transfer the Program, in whole or in part, except as expressly provided herein. You may not sublicense, assign, or otherwise transfer the Program to any third party except by the express written consent of the Developer or Publisher.

TERM
The license is effective until terminated. You may terminate at any time by destroying the Program together with all copies of the Program in your possession. It will also terminate automatically upon failure to comply with any of the terms of this agreement. You agree upon such termination to destroy the Program together with all copies in your possession in any form.

CONFIDENTIALITY OF THE PROGRAM
You understand that the Program is proprietary to the Developer, and agree to maintain the confidentiality of the Program. You agree that neither you, nor any person or entity acting on your behalf, will copy or otherwise transfer the Program, in whole or in part, in any form (including printed source code), to any third party. You agree to retain the Developer's copyright notices, in all forms, throughout the Program. You agree not to de-encrypt or de-compile the Program.

LIMITATION OF LIABILITY
The Program is provided "AS IS" without warranty of any kind, including, but not limited to the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the Program is with you.

In no event will the Developer or Publisher be liable to you for any damages, including any lost profits or other incidental or consequential damages arising out of the use or inability to use the Program, even if advised of the possibility of such damages.

SUPPORT
Support for the Program should be obtained from the Dealer from whom it was purchased. Support pricing and terms are established by the Dealer, not the Developer or Publisher.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT, AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN YOU AND THE DEVELOPER AND PUBLISHER AND IT SUPERSEDES ANY PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATION BETWEEN YOU AND THE DEVELOPER RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

TABLE OF CONTENTS

INTRODUCTION

UnForm is a software product designed to work as a filter between an application and an output device or file, such as a laser printer or a PDF document, or a program like a fax product. Most applications can be simply configured to print through UnForm, which in turn processes the output from the application, determines if custom processing is necessary, and then applies any enhancements before it is output.

For example, if a UNIX program sends output to the spooler like this:

cat *file-name* | lp -dlaser -s 2>/dev/null

Then the output can be changed to use UnForm (note the use of the –oraw option, which can vary by operating system):

cat *file-name* | uf80c -f acct.rul | lp -dlaser –oraw –s 2>/dev/null

Or for better performance, UnForm can print directly from the server:

    cat *file-name* | uf80c –f acct.rul –o ">lp –dlaser -oraw"

UnForm can also work in Windows environments, utilizing TCP/IP printing directly from text print drivers, or via a post-print command line execution controlled by an application program.

UnForm is unique in its ability to analyze report output to determine what, if any, customization to apply. When a report is detected that requires enhancements, UnForm can add line drawing, shading, attributes, font control, and text to the form. UnForm can also handle the processing of multiple copies, multiple output devices, attachments, overlays, and graphic images, and includes support for the complete Business BASIC programming environment to add true programmed intelligence to any form.

The enhanced output can be used to simulate pre-printed forms, or to change the look of plain-paper forms, for which headings and dashed lines are printed by the application, from crude to professional. UnForm can also be used to enhance reports, such as financial statements or aging reports, raising them from mundane to board room quality.

UnForm can produce enhancements on any printer or device that offers the HP PCL5 printer language or PostScript level 2 or 3. This includes most HP LaserJet and compatible printers beginning with the HP LaserJet III, many UNIX faxing software packages, and other products.

UnForm can also produce virtually identical output in Adobe's Portable Document Format (PDF), and similar output in Zebra's ZPL II language, supported on many Zebra thermal label printers. With proper configuration, UnForm can automatically convert its PDF output to any format supported by Ghostscript, including tiff, jpeg, png, and more. Lastly, UnForm can parse column and row oriented reports and produce formatted HTML output.

While UnForm has traditionally accepted plain text print streams and constructed documents from this basic text, version 8.0 adds a new capability to accept PostScript print streams that contain application-formatted documents. In conjunction with GhostScript and the UnForm Windows Support Server, this pre-formatted data is translated to PostScript, PDF, or PCL5 print streams, with optional enhancements such as images, barcodes, text, and drawing features added by UnForm. Further, the text elements provided in the input are available to UnForm jobs for designing full-featured document management applications.

See the UnForm AFO chapter for more details.

CLIENT-SERVER ARCHITECTURE

UnForm utilizes a client-server architecture, where the UnForm processing of documents can occur on a different machine from the application. The resulting enhanced document can be printed, emailed, sent to a fax gateway, or stored at the server, or can be returned to the client machine for printing or storage from its perspective. One important benefit of using a client-server model is that the application process that is sending jobs to UnForm via the client software need not wait for the job to finish if the server will be handling the output. This provides better performance to the application user, particularly for large or complex jobs that take time for UnForm to process.

The UnForm server can run on either UNIX or Windows systems. The server provides the UnForm processing logic and a listener, which handles job requests from clients located on the network.

The UnForm clients can be installed anywhere on the network, on Windows or UNIX systems. On Windows, the client is a native Windows executable. On UNIX, the client is a Perl program, so UNIX systems require Perl level 5.005 or above. Clients perform the application interface work, taking input from the application, submitting it to the server, and in many cases, returning the result back to the client for processing.

There is nothing to prevent the same machine from acting as both client and server, and in fact, the server installation automatically installs a client on that machine. Submitting a job to 'localhost' when the client and server run on the same machine can improve performance, as job data need not be transferred over the network.

For complete information about how to operate the client and server programs, read the Command Line Options chapter. In general, on Windows the server is operated from the Server Manager option or as a Windows service, and on UNIX the server is operated like this:

```
uf80d start
uf80d stop
```

The client supports an extensive set of options. Some simple examples:

```
cat sample1.txt | uf80c –f simple.rul | lp –dhp –oraw
uf80c –i sample1.txt –f simple.rul –o ">lp –dhp –oraw"
uf80c –i sample1.txt –f simple.rul –p pdf –o client:sample1.pdf
```

In the first example, uf80c submits the job and returns the result to its spooler. In the second example, uf80c submits the job and the server prints the result to its spooler. In the third example, uf80c submits the job requesting PDF output, and returns the result to its file sample1.pdf.

SERVER INSTALLATION

UNIX Server download installation instructions:

1. Login as root.

2. Create a directory to hold the UnForm files, and change to that directory.

   Example:
   ```
   umask 0
   mkdir /usr/unform80
   cd /usr/unform80
   ```

3. Uncompress and extract UnForm from the download file.

   ```
   uncompress uf80_xxx_tar.Z
   tar xvf uf80_xxx_tar
   ```

4. Execute the UnForm set up script.

   ```
   ./ufsetup.sh
   ```

   The ufsetup.sh script will create two scripts, called /usr/bin/uf80c and /usr/bin/uf80d. The uf80c program is the client, while uf80d manages the server.

5. Activate demo mode, or activate permanently, using **./license.sh**.

6. Start the server: **uf80d start**

7. Use the **uf80c –v** command to ensure UnForm is installed and set up correctly. The output from this command will display information about the installation. Note that uf80c requires Perl version 5 or higher.

   See the **Licensing** section for activation information.

   Note that you will probably want to place the **uf80d start** command in your system boot scripts, often found in the /etc/init.d directory or a similar location, depending on your version of UNIX.

Windows Server installation instructions:

1. **Windows desktop systems:** Simply execute the downloaded setup executable.

   Windows server systems: 2003: Use Control Panel, Add/Remove Programs, and execute the setup executable. 2008 *without* Terminal Services: right-click the setup executable, and 'Run As Administrator'. 2008 *with* Terminal Services: Use Control Panel, Programs, Install Application on Terminal Server, and execute the setup executable.

   Follow the on-screen prompts from the installer to install UnForm to your system. This will install both the uf80d.exe server program and the uf80c.exe client program. The client program and its associated support files will be installed in the Windows directory, enabling a command line launch without a full path, as the Windows directory is always included in the PATH environment variable.

2. Click the **Server Configuration** option from the Start menu. This will conditionally rename certain files and prompt for several configuration values. The values entered are stored in several local .ini files in the UnForm server directory. You can also use the **Configure** button from within the UnForm Server Manager.

3. Click the **Server Manager** option from the Windows Start, Programs, UnForm 8.0 Server menu.

4. Activate the demo mode, then if desired, activate permanently, by pressing the **Licensing** button and using the form that displays. On line help is available if needed.

5. Click the **Start** button from the Server Manager to start the server manually.

6. Use the **Server Version** option from the Start menu to ensure the server is running properly and the client can operate from the server computer. The output from this command will display the version and licensing information.

7. If desired, you can install the server as a service from within the Other tab in the Configure window. When the UnForm server is run as a service, it is automatically started when Windows boots up. It is generally recommended that the server first be configured and tested while running as an application, and then installed as a service after confirming successful operations.

   The service by default will run under the local SYSTEM account, which typically does not have access to network resources. Use Control Panel, Administrative Tools, Services to modify the service to login under a different account if necessary.

   See the **Licensing** section for activation information.

CLIENT INSTALLATION

The uf80c client software can be used to submit jobs to UnForm from anywhere on your network after the server is installed and operating. The client software is automatically installed on the same machine as the server, so jobs can be submitted locally. However, you can install the client software on any network computer. Any client can talk to any server, so you can mix and match different operating systems as you need. For example, you could install the Windows server, and have both Windows and UNIX clients submit jobs to it.

Clients must be installed on any machine that will be submitting jobs to UnForm. For example, in a Windows network, with the UnForm server installed on a single network server, each workstation that will be submitting jobs must have a client installed and configured to communicate with that server.

The UNIX client is installed from the file uf80c_tar.Z, while the Windows client installer is called uf80c_setup.exe.

The UNIX install steps are as follows:

- Ensure the system has Perl level 5 or higher: **perl –v**
  If not, Perl can be obtained from http://perl.com or http://cpan.org.
- Create a directory for the client, such as **mkdir /usr/lib/sdsi/uf80client**
- Set permissions on that directory: **chmod 777 /usr/lib/sdsi/uf80client**
- Copy the uf80c_tar.Z file to that directory and **cd** to that directory
- Uncompress the file: **uncompress uf80c_tar.Z**. If you have gzip, then the gunzip utility can also uncompress the file.
- Extract the files: **tar xvf uf80c_tar**
- Run the setup script: **./ufcsetup.sh**
- Edit the uf80c.ini file to set up the client configuration:

The uf80c.ini file looks like this:

```
[defaults]
server=localhost
port=27280
#logfile=uf80c.log
#mailto=root
retry=30
wait=2
```

Change the server= line to point to the server host name or IP address, and the port line to the proper listening port configured in the server's uf80d.ini file. The port default is 27280, and will not normally be changed. Note that the server and port can also be specified on the uf80c command line. The values entered here serve as defaults.

If you want uf80c to log errors, uncomment the logfile= line, setting the value to a log file name.

If you want uf80c to email (using the UNIX mail command) error messages to an administrator, uncomment the mailto= line, setting the value to an email address available from the client computer. Note that the Windows client does not support emailing of error messages.

The retry and wait lines set the number of times, and delay between tries, that the client will attempt to connect to the server before giving up. If any retries are needed, and the log file is specified, then a message will be logged.

On Windows, the installation steps are:

- Run the **uf80c_setup.exe** installer program.
- Run the **Configure UnForm Client** option from the Start menu. Enter the appropriate values for the server and port, and optionally the log file.

WEB SCRIPT INSTALLATION

One component of UnForm archiving is a web browser-based interface for browsing, searching, and viewing archived documents, as well as user administration for non-archiving installations. You can interface directly to the UnForm server using its internal HTTP server, which by default listens on port 27282. However, if you wish to use an external web server, you can install the web script as described below.

The web script, a CGI executable, is used to enable Web browser access to library archives when UnForm archiving is used. This script is called uf80a.pl on Unix/Linux, and uf80a.exe on Windows, and is present in the UnForm server installation directory. The script must be copied to a location where the local web server can execute scripts. Once the script is copied and configured, a Web browser can access it with the appropriate path, such as:

```
http://myserver/cgi-bin/uf80a.pl
http://myserver/scripts/uf80a.exe
```

Note the name of the script can be changed however appropriate.

Unix or Linux
If the web server is Apache, then there is generally a cgi-bin directory that can be used to host the uf80a.pl script. It is also common to configure additional directories to support CGI scripts, so the web server administrator may chose a different location. Alternatively, sometimes file extensions are mapped to always execute as CGI scripts (.cgi for example), in which case uf80a.pl can be renamed to uf80a.cgi and placed in any directory accessible to the web server.

Once the script has been copied to the correct location, verify that the script signature line (line one) properly references the location of Perl on the system. The default is: #!/usr/bin/perl, but another common location is #!/usr/local/bin/perl. Note the "#!" prefix is part of the syntax and is required.

Windows
Most IIS installations include a directory c:\inetpub\scripts, and this is a suitable place for uf80a.exe. Alternatively, the web server administrator can define virtual directories that support scripting, and uf80a.exe can be copied there.

Configuration
If the UnForm server is running on the same machine as the web server, and is listening on the default port of 27280, then there is no configuration necessary. However, a file uf80a.ini, in the same path as the uf80a script, can be defined with two lines in it to override these assumptions:

server=*server name or IP address*
port=*listening port*

Performance of the web interface is best if the web server and UnForm server are on the same physical machine. If different machines are used, firewall configuration may be necessary to enable the web server to connect to the UnForm server's listening port.

Help Files
The help files for the browser interface are stored in the web/*language*/help directory under the UnForm server (the language defaults to en-us). These files are not accessible through the CGI interface, so they must be copied to a location on the web server to be accessible (alternatively, an alias or virtual directory can be configured to point to this location if the web server resides on the same system as the UnForm server). A web server administrator must make these help files available, and then the helppath=*url* line can be updated in the [archive] section of uf80d.ini. The help path configured is used as the prefix to the index.html file of the help system in all the Help links in the browser interface.

For example, in Apache, a virtual directory /archelp might be configured like this:

```
Alias /archelp/en-us/ /usr/lib/unform8/web/en-us/help/
```

Then the help path could be configured like this:

```
helppath=http://[HTTP_HOST]/archelp/[LANGUAGE]
```

CONFIGURING THE SERVER

The server is configured via the uf80d.ini file, which can be edited with any text editor. On Windows, many of these options can be configured with the Configure button in the Server Manager. In addition to these items, you can also configure access to Ghostscript, Image Magick, or Image Alchemy elsewhere in the uf80d.ini file. See the Configuring External Programs chapter for more details.

In the defaults and security sections, here are the values available:

[defaults] section	
port=*n*	Sets the primary listing TCP/IP port to *n*. The default is 27280. Note that if you use NAT translation or if you have a firewall between the clients and server, then this port (along with the procports defined below) must be configured to allow clients access.
logfile=*path*	Sets the name of the server's log file to *path*. By default, it is stored in the UnForm directory. Standard log entries include connection information. Detailed logging includes verbose data transactions.
logdetail=*n*	Set *n* to 0 for standard logging, 1 for detailed logging. You should not leave detailed logging enabled for normal use, as the log file can grow very large.
timeout=*n*	Set *n* to the number of seconds that a connection can remain idle before closing. The default value is 3600, or one hour. Setting this value to 0 will avoid timeout-based disconnects. This value primarily affects designer connections, which can remain active for long periods.
age=*days*	This value sets the maximum age, in days, of job log entries. When jobs are submitted, basic job information is kept in a log file. If errors were recorded, the error file also remains in the temp directory under the UnForm server. After this many days, the files and log entries are automatically removed. A fraction of a day can be supplied, such as age=.25 for 6 hours.
agetmp=*hours*	This value sets the maximum age, in hours, of ./temp/tmp files, which is the default directory for work files.
rulefile=*path*	Sets the default rule file to *path*, used for jobs that do not specify a rule file on the command line.
bbpath=*path*	If the bbxread() function is used, this value points to the BBx executable that is invoked when required, such as /usr/lib/basis/pro5/pro5.
library=*path1*;*path2*;…	Sets directory paths that are automatically searched for rule files, images, and attachments. By default, UnForm searches the UnForm directory and also supports full paths.
sshost=*host*	Sets the default host IP address or name of the Windows Support Server. In addition, the **sshost()** function can be used in a code

	block to specify the host and port at run-time.
ssport=*port*	Sets the default port to connect to the Windows Support Server on the host specified by sshost.
imageage=*days*	Images that are converted by an external conversion program or by the Windows Support Server are cached by default. The last date an image is used is also stored, and images that have not been used in *days* days are removed automatically.
stylesheet=*name*	Sets the name of the style sheet used by the archive browser interface programs. A file called "default.css" is provided with the server installation (found in the web/en-us directory). This style sheet is also used when archives are exported to static HTML structures.
bufsize=*bytes*	An initial block is tested for each job in order to determine if the job contains binary data or text data. The size of this block defaults to 8196 bytes, but you can adjust it to any integer value with this entry. The minimum value is 1024.
cr=0|1|2|3	Controls default handling for embedded carriage return (chr(13)) characters in lines read from the input stream. This value may be overridden with the –cr command line option. • 0 will truncate lines at the first CR. • 1 will strip CR character, so the line continues as if the character were not present. • 2 will fold lines, and non-space characters are placed in the line buffer, simulating an overstrike. • 3 will fold lines and insert an extra space, which accommodates Windows Generic/Text Only printers that overstrike conflicting characters.
repair=0|1	If set to 1, the next start of the uf80d server will attempt to repair certain control files, such as the job history database and user table. Use this feature if you suspect corruption in one of these files. It should normally be set to 0.
tcpportretry=*n*	The number of times a job received on a direct TCP/IP port will be retried if a non-license (998) error occurs. As an example, if a network printer goes down and UnForm returns errors trying to open the output device (-o *devicename*), this sets the maximum number of times the job will be submitted by the port sweeper. The sweeper runs each time a job is submitted and every 5 seconds when idle. Setting this value to a reasonable number allows for temporary problems to be self-corrected without causing an unlimited number of log and error files to build up due to a configuration issue. To release jobs once a problem is corrected, manually remove the *.rty file(s) from the rpq directory.

pdftrans=0|1	Sets the default PDF transparency setting. If 1, then PDF files will use transparency.
textjob=0|1	Sets the default behavior on generation of the textjob$[all] array, which is a collection of all print lines for the job. This array can be useful when performing report mining operations, or parsing a full job into pages in a prejob code block, but when large print streams are processed, a significant amount of memory and CPU resources are consumed generating the array. The -textjob, -notextjob command line options override this setting.

errnotify=*email address*
errnotifysubject=*subject*
	If an error occurs while running a job, messages are written to a job number error file (temp/*jobno*.err). This file remains on disk for a configured amount of time, typically seven days. You can configure an email address (or multiple emails separated by commas) to have the server send the contents of these error files to an administrator, reducing the need to proactively monitor for errors.
	
	Only jobs that encounter runtime errors trigger mailing. Errors in jobs being tested in the design tool are not sent. Also, some errors can occur only on the client side, such as an invalid server address. In those cases, the server is unaware of the error and no message will be sent. Such errors need to be captured via client error handling.
	
	The *subject* specified may include a tag "@jobid" to reference the job number, which can be helpful to cross reference back to a rule file and rule set by locating the "Job complete" message in the server log file.
	
	If an error occurs while emailing, a message is logged in the server's uf80d.log file. This feature depends on having email properly configured (in prog/mailcall.in or the [mailcall] section of uf80d.ini), and is available starting in version 8.0.28.

[security] section	
allow=*list*	This is a semi-colon delimited list of valid IP addresses or wildcards that are allowed to connect to the server. Note that the loopback address 127.0.0.1 is always allowed to connect. The default list is 192.*.*.*;10.*.*.*, which allows the two standard non-routable LAN spaces to work.
designer=0|1	If set to 1, the Design Tool will require a login and will encrypt rule files when saved. The login is restricted to a user who is either an administrator or has been granted Design Tool access via the user maintenance features of the archive web browser interface. Rule files saved or published in encrypted format can be read by any UnForm 8.0 server, regardless of its designer security setting, but

	can only be edited by the design tool. If set to 0, the Design Tool does not require a login, and rule files are maintained as text files, which can be edited using any text editor as well as the design tool.
[tcpports] section	
port=*options*	Each line defines a port on which the server listens for raw print job deliveries, such as from Windows TCP/IP ports. Each job submission is then processed using a uf80c command line configured with a pre-defined -ix option plus any other *options* defined. For more information, see the TCP/IP Monitor chapter.
[archive] section	
deflib=*defaultlib*	Sets the default library name, for use when archive commands do not specify a library name. This library will be placed under the default “arc” subdirectory below the UnForm server.
keywords=*n*	Specifies the maximum number of default keywords generated for UnForm job-based archives. Default keywords are unique words generated from the job input stream that do not match patterns defined in the nonwords= file. If this value is set to -1, then all unique words become keywords. The benefit of this is that more words of job print streams are available for searching. The cost is greater time spent parsing reports for words and additional disk space utilization.
nonwords=*file*	Specifies a file which contains lines of regular expressions for “words” that should not become keywords. See ufnonwords.txt for examples.
nonchars=*charlist*	This is a list of characters that are removed from keywords. The default list provided with UnForm is: <>{}[]()*=~`'"+|
endchars=*charlist*	This is a list of characters that are removed from the end of keywords. For example, you may want to remove periods from the ends of words as a period typically ends a sentence. The default list provided with UnForm is: .?!,;.
searchage=*days*	When archive searches are performed in the Web browser interface, work files are generated. This sets the maximum number of days these files will remain on disk.
webdirs=*dir1;dir2;…*	If you need to support multiple languages, or you wish to offer a customized user interface for archive browser users, you can copy the ./web/en-us directory to other ./web/* directories and customize them. In particular, the messages.txt file and various html templates or style sheets can be customized. This directory list (and associated name=*title* values in each messages.txt file) are presented in the browser login screen.
sesage=*hours*	Set the number of hours a browser session can last before a login is required again. By setting this to 0, browser users must login each time their web browser is re-started and the web interface is

	accessed. Set it to a large number to allow users to login once per workstation and have that login remembered.
defperm=*perms*	Sets the default permissions on new libraries. Set to zero or more semi-colon delimited letters, r, w, and d for read, write, and delete. For example, defperm=r;w for default read and write, or defperm=r for just read only, or defperm= for no default permission, meaning only administrator logins can access the library initially.
defseq=0 or 1	Sets the default Force Sequence on Sub ID flag value for new libraries. If set to 1, then sub ID's are auto-sequenced to prevent overwriting.
pdfname=*name*.*.pdf	In the browser interface, when images are consolidated into a single PDF, a file name is suggested when the PDF file is saved or an attachment is emailed. This value forms a pattern, with an asterisk positioned to indicate where unique sequencing can be applied. Use this to identify a company, such as SDSI.Document.*.pdf, so when an email recipient receives an email, the attachment name will be readily identifiable.
dtdel=http://*unformserver*:*port*	If this line is enabled, then the browser interface will display a menu entry to launch the desktop delivery browser client.
selfmanage=0|1	If set to 1, the browser interface will allow users to email their login and password to themselves, and can change their own password.

ses_wlst=0|1
ses_notext=0|1
ses_mailfrom=*email*
ses_tiftopdf=0|1
ses_imgtopdf=0|1
	These entries provide session defaults for the browser interface. The ses_wlst entry establishes the default for image details (wide listings) when browsing or searching. The ses_notext can be used to turn off @text images when browsing. ses_mailfrom provides a default From address when emailing consolidated documents. ses_tiftopdf and ses_imgtopdf enable TIF or all images to be converted to PDF before viewing in the browser. This capability requires that Image Magick be configured on the UnForm server or in the Windows Support Server.

logo=*filename*	Sets the logo file name used for most pages in the browser interface. This should be a file format that can display as an inline image in browsers, such as jpg or gif, and should be a small file to avoid page formatting issues. The default logo is a simple UnForm icon, unform.gif.
brhistcnt=*count*	Sets the maximum number of browse history items that are maintained for a user. These keep track of recent browse libraries, orders, and starting points to assist a user who frequently uses specific browse functionality.

helppath=*url*
helppath2=*url*
	Specifies the URL entry points for the browser interface help directories (help is for users, help2 is for administrators). When using the internal web server, this defaults to a local path web/*language*/help. However, if browser access is configured through an external web server using the CGI script, the help directory must be configured on that web server and the helppath

	URL modified. You can use [HTTP_HOST] and [LANGUAGE] tags in this path. Below is the structure used for the internal web server:
	
	helppath=http://[HTTP_HOST]/web/[LANGUAGE]/help
	helppath2=http://[HTTP_HOST]/web/[LANGUAGE]/help2

enablefax=0|1	Enable fax tabs in the browser interface. Fax submissions rely on the deliver() function, so faxing must be configured in the deliver.ini file for this functionality to work.
faxcover=*name1*[,*name2…*]	When faxing is enabled, this provides a list of one or more cover page names that are acceptable for use with the deliver command's configured faxing product. For example, msfax users could rely on the "generic" cover definition by specifying faxcover=generic. The browser interface offers a choice of no cover, plus any cover names defined here. Multiple names are delimited with commas.
[mailcall] section	
name=*value*	The mailcall section can be used to define mailcall values not available in the email command and email() code block function, such as timeout or bodymime, or to provide a default setting if the command or function doesn't supply a value (or supplies a null value). Any options not set in an email command or function will be filled in with values in this section. For example, if you want a bcc sent to a local support account by default, add a line that says bcc=*email address*. Or, if you find that the default timeout of 30 seconds isn't enough time for a slow internet connection, add a line like timeout=60.
[httpd] section	
port=*port*	The desktop delivery server is based on the HTTP protocol. The port on which it listens defaults to 27282, but that can be changed by modifying this entry. Note this server can also be used to access archive libraries, using http://*server*:*port*/arc.
logdetail=0|1	If set to 1, the desktop delivery server will log details into the server's log file, usually uf80d.log.

block=*value*
cmpminsize=*value*
minhelpers=*value*
filecache=0|1
	These values can be tuned to adjust the desktop delivery server's performance.

Note also many parameters are stored in the ufparam.txt file. You can create a custom version of this file, called ufparam.txc, which will be used instead of ufparam.txt. Any new parameters that are added during a release cycle are documented in the readme.txt file, and can be added manually to keep ufparam.txc up to date if necessary.

Of particular interest in ufparam.txc is the font configuration. All fonts are assigned a numeric ID. Those that are common in pcl5 printers have pre-assigned values from HP, while soft fonts can be given user-defined numeric IDs. These name=number pairs are defined in the [fonts] section. ID numbers can then be assigned to soft font names in the [psmap] and [ttmap] sections, or mapped to PDF base fonts in the [pdfmap] section. See the standard ufparam.txt file for examples and notes.

When UnForm processes a text or font command, it attempts to match a named option with a font name in the [fonts] section, and it then uses the associated font number. Alternatively, the text or font command can identify the font number directly, with a font *n* option.

Once a number is identified, UnForm then looks for a native soft font definition, depending on the output format. It looks in the [softfont] section for pcl5 output, or the [psmap] section for postscript output, or the [pdfmap] section for mapping to an internal PDF base font. If no match is found, then the [ttmap] section is scanned for a match, and the associated TrueType soft font is embedded in the output and used.

An example of mapping a True Type font would look like this:

```
[fonts]
…
vera=19200

[ttmap]
19200=Vera,VeraBd,VeraIt,VeraBI
```

In the [fonts] section is a *name*=*number* pair. The *number* is user-defined and must not conflict with the various fixed PCL font numbers found in the section. The [ttmap] section contains a number=font(s), where a list of font file names (without the .ttf extension) is provided for normal, bold, italic, and bold-italic versions of the font. Note that not all fonts provide all these versions. True Type font files are found in the ./ttfont directory, or on Windows in the %windir%\fonts directory, or can be specified as full path names.

CONFIGURING EXTERNAL PROGRAMS

The UnForm server supports the use of three external programs for handling two tasks: image scaling and conversion, and document imaging conversion.

For image scaling, you can configure either Image Alchemy, a commercial product available from Handmade Software (http://handmadesw.com), or Image Magick, an open source product available from http://www.imagemagick.org. Once configured, image scaling is automatically used when an image command contains size information and the image file is not a native file for the output format UnForm is generating.

For document image conversion, you can configure Ghostscript, an open source or commercial product available from http://ghostscript.com. Document imaging is managed by the –p command line argument, and it enables a series of additional drivers, such as tiff, postscript, and png. Ghostscript is also used internally by UnForm when PDF files need to be converted to other formats.

Once the appropriate programs are installed, edit the uf80d.ini file to configure them.

Use the [images] section to configure Image Alchemy or Image Magick, first by defining a converter=*path* entry, where *path* is the execution path of the alchemy or convert programs. If the path is in the operating system's PATH variable, then just a simple name will be required. Since the server, uf80d, will be executing the program, you should make sure that the user under which it runs includes the proper environment variable definitions.

In addition to the executable, define several command line argument lines for pcl, pclc, and pdf, and optionally others that can be called out by the option item of the image command. Generally, you can simply uncomment the proper lines for Alchemy or Magick. The pcl command is invoked for laser output, and the pdf command is invoked for PDF output. If the image command's color option is used, or the –color command line option is used, then the pclc command is invoked. Below is a sample uf80d.ini [images] section, with Image Magick enabled:

The following substitions are made at runtime:

%i	for the input image file
%o	for the output image file that UnForm will use
%d	for resolution in dots per inch
%x	for image width in pixels
%y	for image height in pixels
%g	gamma value from the image command

```
[images]
# External image conversion/scaling program setup
# 1) Define program path: converter=pathname
# Use a full path if necessary, as this becomes a system call in UnForm.
# On Windows, this will very likely be necessary.

# 2) Define arguments to be passed to converter for pcl, pclc, and pdf.
# Use %i for input image, %o for output, %d for dpi, %x for width, %y for height
# pdf should not contain %x/%y, as scaling is performed by Acrobat.

# Options passed from image command line can be appended to the name with a dash.
# i.e. image 10,10,10,10,"image.bmp",option 123 would use pcl-123 or PDF-123.
# Options can be up to 10 characters long, and are case sensitive.

# Examples for Image Alchemy:
#converter=alchemy
#pcl="%i" "%o" -o -Q -D %d %d -+ -Xc%x -Yc%y -P 103 >/dev/null 2>&1
#pclc="%i" "%o" -o -Q -D %d %d -+ -Xc%x -Yc%y --r 9 >/dev/null 2>&1
#PDF="%i" "%o" -o -Q -D %d %d --d -8 >/dev/null 2>&1


# Examples for ImageMagick:
converter=convert
pclc="%i" -density %dx%d -colors 256 -dither -resize %xx%y "%o" >/dev/null 2>&1
pcl="%i" -density %dx%d -monochrome -resize %xx%y "%o" >/dev/null 2>&1
PDF="%i" -density 300x300 -colors 256 "%o" >/dev/null 2>&1
#PDF-72="%i" -density 72x72 -colors 256 "%o" >/dev/null 2>&1

# PDF-72, above, is a 72 dpi image conversion, and would be specified
# with 'option 72' in an image command.  The resulting file will be much
# smaller than the 300 dpi image shown in PDF=, though quality may suffer
# too much for use, depending on the image itself.
```

Use the [drivers] section to define the Ghostscript-hosted imaging drivers. When this feature is enabled, the –p *driver* option supports a series of new names, all derived from an intermediate PDF document that is converted at the end of the job to the specified format. First, enable the gs=*path* line to instruct UnForm how to run Ghostscript. On UNIX, this is often just the word "gs", while on Windows it is often a full path to the gswin32c.exe program.

The “pdffitpage=*n*” option is used to indicate if this version of GhostScript supports the –dPDFFitPage option, which was added to GhostScript at version 8.10. If this value is 1, then UnForm can optimize management of PDF files added to a job using the **images** command. Without this capability, all images must be converted to full page sizes and then scaled. Note that an alternative to using a server copy of GhostScript is to set up a Windows Support Server and execute GhostScript on that machine. This enables sites running older versions of Unix or Linux to access current versions of GhostScript. To force the use of the Windows Support Server, disable the gs=*path* line (#gs) and use the sshost setting or code block command to enable use of the Support Server.

Other entries are simply *name*=*device*,*multipage*,*dpi*, where *name* is the UnForm driver name, *device* is the –sDEVICE name used by Ghostscript, *multipage* is a 0 or 1, where 1 means the output is multi-page=multi-file and 0 means all pages go to a single file, and *dpi* is the dots-per-inch resolution.

Note that the use of multi- or single-page output is often dependent on the image format. For example, bmp files do not support multiple pages per file, while tiff files do.

Note that the graphical designer may rely on the png entry shown, depending on how it is configured.

```
[drivers]
# enable Ghostscript drivers by uncommenting the gs= line
gs=gs
# windows would typically need a full path
# gs=c:\gs\gs8.xx\bin\gswin32c.exe
pdffitpage=0
# driver lines are structured as name=gsdevice,multipage,density
# gsdevice is the Ghostscript sDEVICE value
# multipage is Boolean 0 or 1, 1 means -o file is file<page>.ext
#    Many formats require a 1, as the image format supports only a
#    single image per file.
# density is output density, as hhh[xvvv] (horizontalxvertical) dpi

bmp=bmp256,1,300
bmpmono=bmpmono,1,300

tif=tiffcrle,0,300
tifmono=tiffg3,0,300

png=png256,1,300
pngmono=pngmono,1,300

jpeg=jpeg,1,300

ps=pswrite,0,300
eps=epswrite,1,300
deskjet=deskjet,0,300
```

DELIVER CONFIGURATION

The deliver.ini file contains configuration information for the deliver command and deliver() code block function. The file contains a [default] section, an [email] section, and one or more fax definition sections. The [default] section contains a fax= line that specifies which fax definition section is used. It also contains logging parameters.

A fax definition indicates the type of interaction that is required, command line, email (SMTP), or the internal msfax option, which utilizes the Windows Support Server to send faxes via Microsoft Fax. Each configuration line can contain optional and required tags, which are substituted at runtime with tag values specified in the deliver command or deliver() code block function. Tags are used to specify fax numbers, subject lines, recipient names, and other values, and the values from the commands are mapped to the configuration each time a delivery is performed.

There are five pre-defined tag names that can be used in the configuration.

Tag	**Usage**
%to	Fax number or email address.
%faxfile	File to be sent (the output from UnForm, typically of a subjob). The deliver command manages this value automatically. The deliver() function has a file name argument that this tag references.
%errfile	An error file, which if used and contains content will be returned in the errmsg$ argument of the deliver() function. UnForm generates the file name automatically, and then expects to find error messages, if any, in the file when the fax or email is submitted.
%rspfile	A response file, which if used and contains content will be returned in the response$ argument of the deliver() function. UnForm generates the file name automatically, and then expects to find messages, if any, in the file when the fax or email is submitted. For example, this might be the standard output of a vfx command line, which can contain a fax job number.
%attach	Used for additional documents beyond the %f file. All attach tags are accumulated, with space delimiters for command methods, and comma delimiters for email methods, then the entire list is substituted for the %attach marker. The msfax method does not support attachments.

Additional tags are referenced with a %*name* syntax, and are substituted from the tag options provided by the deliver command or deliver() function. Optional tags are specified using a { %*name* } syntax, where a tag and any text can be specified inside the curly braces. After all %*name* substitutions have occurred, any remaining {…} sequences that contain %*name* tags are removed before the delivery is actually executed.

When the deliver command generates a document for delivery, it does so with an UnForm subjob. When delivering to an email address, the subjob always produces a pdf file. When delivering to a fax number, the type of file is configured with the format=pcl|pdf|ps line. When executing the subjob, several command line options are automatically managed. The fax configuration section can specify additional options, as can the deliver command itself.

Below are commented examples of sections:

Default section
This example [default] section specifies that the vsifax fax definition will be used for faxing. Daily delivery log files will be placed in the deliver subdirectory under the UnForm server, tags will be logged in addition to basic delivery data, and log files will be retained for 30 days.

```
[default]
 fax=vsifax
#fax=msfax
logdir=./deliver
logtags=1
logage=30
```

Email section
This [email] section defines a To: header format with an optional quoted name and the to address. The name can be supplied via a name="*value*" tag in the deliver command or function. If not supplied, then the To: header will just have the email address, automatically supplied as the "to" tag.

Other values can be supplied via additional tags. The email will always include a file attachment, as generated by the deliver command or supplied to the deliver function. In addition, if there are any "attach *filename*" tags, those files will be attached to the email as well.

```
[email]
to={"%name" }%to
subject={%subject}
msgtxt={%note}
attach=%f{,%attach}
logfile={%logfile}
#otherhead=
#login=
#password=
#server=
# Note, additional options come from [mailcall] section in uf80d.ini,
# or the mailcall.ini file.
```

Vsifax section
The vsifax section defines a fax gateway that uses the vfx command to send a fax. The method is set to "command", indicating that a command line is created and executed. The command is specified over several lines in the deliver.ini file by using \ at the ends of lines to indicate that the line continues on the next physical line.

There are several optional elements, such as name and subject. If they are supplied as tags in the deliver command or deliver() function, then -t options will be added to the vfx command line. If not, then those elements are suppressed from the command line. For example, if one tag was 'name="John Smith"', then there would be a '-t tnm="John Smith"' added to the command line.

Standard output is captured in a response file, and error output is capture in an error file. The use of %r and %e in the command allows the deliver() function to return the command's response or error message.

When the deliver command executes the subjob to produce the file to be faxed, it will produce it in pcl format (using a -p pcl option). In addition, a -nohpgl option will be used.

```
[vsifax]
method=command
command=vfx -n '%to' -F pcl \
 {-t tnm="%toname"} \
 {-t tco="%tocompany"} \
 {-t sub="%subject"} \
 {-t fnm="%fromname"} \
 {-t fco="%fromcompany"} \
 {-t ntx="%note"} \
"%faxfile"{ %attach} \
 >%rspfile 2>%errfile
format=pcl
options=-nohpgl
```

Rapidfax section
The rapidfax section shows an example of using a third-party Internet faxing service for sending faxes. Such services accept email submissions via email. PDF format is generally supported, and many services offer support for additional document types, meaning you can add additional attachments, such as text files, image files, or Microsoft Office files.

The method is specified as "email" to indicate that UnForm will submit faxes by emailing them to the fax service. The To address for most Internet services is *faxnum@service*.com. In the case of Rapidfax, the domain is rapidfax.com. Rapidfax expects fax submissions from a specific email address, and with a user-configured subject line. In this example, the From address and subject lines are hardcoded. There can be a message body, specified by a note tag, and both the submitted file and any other files specified by attach=*filename* tags are submitted for faxing.

```
[rapidfax]
method=email
to=%to@rapidfax.com
from=you@yourcompany.com
subject=yourpassword
msgtxt=%note
format=pdf
```

```
attach=%faxfile{,%attach}
```

Msfax Section

The msfax section specifies the "msfax" method, which is an internal UnForm method that works in conjunction with the Windows Support Server. The command value specifies tag names documented in the Windows Support Server chapter and the msfax() function.

```
[msfax]
method=msfax
format=pdf
command={toname="%toname %tocompany"}\
 {,fromname="%fromname"}{,fromcompany="%fromcompany"}\
 {,subject="%subject"}{,note="%note"}{,cover="%coverpage"}
```

BROWSER FORMS CONFIGURATION

The archive browser interface supports custom forms that can be accessed from specific points. These forms, when submitted, execute an UnForm job and provide that job with information from the form as well as the user's browser session and the CGI environment. The UnForm job can perform tasks that might involve updating document properties on a selected or related document, or execute some other task related to a document.

When the job is run, it can be designed to return data to the browser, in the form of a PDF file or HTML text. Optionally, the job can be run asynchronously and a configurable response message is returned instead.

The job must be designed to operate with no usable input stream. If the job's output is to be returned to the browser, the first page of data will be replaced or suppressed by the job. Variables are provided to access information the user completed in the form, so typically the job will be designed with code blocks that interact with this data.

There are several related configuration steps for setting up browser forms:

- HTML forms must be defined and stored in the server's active web/en-us or web/*language* directory. The forms must have a .html extension.

- A rule file and rule set must be created, designed to act upon the data from the form. This rule set will have access to some template variables: cgi$, env$, and session$, which are populated from the form and the user's session.

- The forms.ini file must be defined to enable users or groups to access forms designed for a given library and document type.

HTML Form Structure

The HTML form may have any valid HTML structure for a form, but must contain a few mandatory elements. The form_sample.html file found in web/en-us provides this example form code:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
    <link rel="stylesheet" type="text/css" href="[SCRIPT_NAME]?fl=default.css" />
    <link rel="stylesheet" type="text/css" href="[SCRIPT_NAME]?fl=custom.css" />
    <script language="JavaScript" src="[SCRIPT_NAME]?fl=common.js"></script>
    <title>
      Document Approval
    </title>
```

```
</head>
<body onunload="saveWindowInfo()">

<h2>Approve or Deny Action on this document</h2>

<form name="approval" action="[SCRIPT_NAME]" method="post">
<input type="hidden" name="a" value="uf">
<input type="hidden" name="args" value="-f cgiforms.rul -r approval -p pdf">
<input type="hidden" name="run" value="">
<input type="hidden" name="async" value="0">
<input type="hidden" name="lb" value="[lb]">
<input type="hidden" name="doctype" value="[doctype]">
<input type="hidden" name="docid" value="[docid]">

<table class="form">
      <tr>
            <th>Library</th>
            <td>[lb]</td>
      </tr>
      <tr>
            <th>Doc Type</th>
            <td>[doctype]</td>
      </tr>
      <tr>
            <th>Doc ID</th>
            <td>[docid]</td>
      </tr>
      <tr>
            <th>Categories</th>
            <td>
                  <script>
                        // present categories in lines
                        var cat='[categories]';
                        cat=cat.replace(/;/g,"<br>");
                        cat=cat.replace(/\|/g," &gt; ");
                        document.write(cat);

                  </script>
            </td>
      </tr>
      <tr>
            <th>Approval</th>
            <td>
                  <input type="radio" name="approval" value="0" checked>Denied<br>
                  <input type="radio" name="approval" value="1">Approved<br>
            </td>
      </tr>
      <tr>
            <td colspan="2" style="text-align:center;">
                  <input type="button" value="Submit" onclick="submitForm()">
                  &nbsp;&nbsp;&nbsp;
                  <input type="button" value="Cancel" onclick="window.close();">
            </td>
      </tr>
</table>
</form>

</body>
</html>
<script>
      function submitForm() {
            document.approval.run.value="y";
            document.approval.submit();
        }
</script>
```

This example shows a form related to particular document, identified by the CGI fields lb, doctype, and docid. Document field values are represented by tags **[*fieldname*]**, where the field name can be any of lb, doctype, docid, date, time, title, entityid, notes, keywords, categories, links, dateupdated, and timeupdated. Date fields are provided in YYYYMMDD format, time fields in HH:MM:SS format, keywords and links in semicolon-delimited format, categories in semicolon (category), vertical bar (segment) delimited format, and notes with hard line breaks represented as "\n" literal sequences.

In the form above, notice how javascript code is used to display categories with different line and segment delimiters.

Several fields must submitted with the form in addition to lb, doctype, and docid. These include **a** (set to "uf"), **args** (set to a string of command line arguments), and **run** (set to a non-null value when the form is submitted (see the submitForm() function at the bottom of the document). Optionally, the **async** field can be set to 1 to run the UnForm job asynchronously. When this happens, the browser response is based on the form_async.html document found in web/en-us (or web/*language* if configured). Otherwise, the browser response is either the UnForm job output (typically a PDF file) or a text response set into the cgiresponse$ variable in one of the job's code blocks.

It is possible to have the text response contain script code that simply closes the window:

    <script>window.close();</script>

The variable **args** is especially important, as it specifies, at a minimum, the rule file and rule set to execute when the form is submitted. It may also supply other options, such as -prm parameters, or a -p pdf if the job's output will be the response to the browser.

CGI-Driven Rule Sets

The UnForm job that runs is a rule set named in the args variable of the HTML form, with the -f *rulefile* -r *ruleset* options. A sample rule set is defined in the samples/cgiforms.rul file, called "approval".

```
[approval]
prejob{
        approved=num(cgi.approval$)
        cgiresponse$="Approved"
        if not(approved) then cgiresponse$="Not approved"
}
```

In this very simple rule set, a value is sent from the HTML form field "approval". This field is a radio button with a value of 0 or 1, depending on which button is checked (see the HTML sample above for the HTML form coding used). The variable cgiresponse$ is set to a text value that is returned to the browser.

Alternatively, if cgiresponse$ is not set, then the job's output is returned to the browser. In this case, the HTML form would specify a "-r approval2" and "-p pdf" in the args variable. The result will be a simple PDF document with text indicating the value of the approval field. Note that the print stream data contains data communicated to the rule set that is normally of no use to the user, hence the use of the notext command to suppress print stream text. Other techniques might be code block resets of the text$[] array, or an erase command.

```
[approval2]
prejob{
     approved=num(cgi.approval$)
}
notext
text 10,10,{"Approval: "+str(approved)},cgtimes,12
```

There are three template strings provided to code blocks in a CGI-driven rule set:

- cgi$ contains all the form fields that are present in the HTML form. All cgi$ fields are string fields, so all are accessed as above: cgi.*field*$.

- cgienv$ contains the CGI environment variables, if needed, such as cgienv.script_name$ and cgienv.remote_host$. CGI environment variables are documented in many places on the Internet and in numerous books on web scripting.

- cgisession$ contains fields related to the user's session. Of particular note might be the user login ID, which is found in cgisession.s_userid$.

Javascript Execution of Rule Sets

A function is supplied in the common.js Javascript library that is loaded into web forms (note the header line containing fl=common.js). This function executes a rule set on the server, and the rule set's cgiresponse$ value is returned to Javascript.

The syntax of this function is:

Var txt=runRuleSet("*rulefile*","*ruleset*",*flds*[,*args*])

The rulefile and ruleset arguments are self-explanatory, as they simply become –f and –r arguments to an UnForm job. The *flds* argument can be either a URL=encoded string, such as "name=Jim%20Smith", or an JavaScript object containing name:value pairs, in which case the function will URL-encode all the names and values. The fields defined in this argument are available in the rule set as cgi.*name*$. If additional job command line arguments are required, they can be supplied in the optional args string argument.

Note if there is data in delimited or INI format available on the server and needed in Javascript, there is a json object that can be used in the rule set to convert such data into JSON format, enabling easier and faster access in the script code.

Form Access Configuration

The forms.ini file contains configuration information that the browser interface uses to provide automatic links to forms under certain circumstances. A sample forms.sds file is provided with the UnForm installation for reference.

Document-Oriented Forms
When a user is logged in and viewing document properties or a document image, a Forms tab is offered if any forms are configured that meet the session criteria, and links to all available forms are provided in that panel.

Here is a sample forms.ini file:

```
 #sections by library|doctype
# user:<name>=form name[,description]
# group:<name>=form name[,description]
# *=form name[,description]

[demo_sales|OpInvoice]
user:theboss=form_approval,Approval Form
group:sales=form_rep_action,Sales Rep Action
*=form_site,Site Form Info
```

Sections in this file are identified by a [*library*|*doctype*] heading. When a document of that document type is viewed from that library, forms configured on the following lines will be available, depending on the user and user group membership. Any given form is only presented once, even if the circumstances would cause multiple configuration lines to present the same form.

In the above example, the user "theboss" can access the form_approval.html form, any user in the group "sales" can access the "form_rep_action.html" form, and any user can access the "form_site.html" form. All forms that meet the security criteria are available. For example, user "theboss" will have access to form_approval and form_site, and any user in the group "sales" will have access to form_rep_action and form_site.

Library-oriented Forms
Library-oriented forms are available in the Browse interface when a library has been selected. Sections for these forms are identified with the header [*library-name*|*].

When library forms are invoked, the doctype and docid fields are set to "*".

Search- and Marked-Related Forms
In addition to forms selected when viewing documents of a configured library and document type, forms can also be executed from search results and marked record lists. These are configured in a similar manner, with sections in the forms.ini file.

A [search] section denotes user or group form access when viewing search results. If the results of a saved search are being viewed, then a section named [search|*saved search name*] can be defined, also with user:*userid* and/or group:*groupid* access. Search-based rule sets can use a search object's doclist object to navigate and react to the search results.

Finally, a [marked] section can be defined to allow forms driven from a marked records list. The marked object can be used to navigate and manipulate the marked record list for a user's session.

```
[search]
group:managers=form_transfer,Transfer Documents Form
user:admin=form_transfer,Transfer Documents Form

[search|CurrentInvoices]
group:AR=form_CurrentInvReport,Report of Current Invoices

[marked]
user:jsmith=form_jsmith_purge,Purge Marked Documents
```

Menu-Related Forms
Beginning with version 8.0.26, a new forms.ini structure can be used to add custom panels to the main menu of the browser interface. Panels are added to the bottom of the menu, and link to web forms in the same window or a new window. A URL is constructed based on configuration entries, so that when the user clicks the panel, the custom web form is presented. The code of this form can perform whatever tasks are necessary, even to the point of loading a new page if desired.

Menu panel configuration requires multiple sections. First, a [menu] section must be defined that specifies form names and default titles for different users or groups. This configuration is almost identical to other web form sections, and the active forms are selected based on the session user. One exception to this format is that a ~*value* suffix can be added to the form name to allow multiple sections to be configured for a given form. The suffix is used only to locate a related configuration section, and is not considered part of the form's related HTML file name.

For each menu web form name, an additional section must be configured in the file to describe the attributes of that panel and how it launches the web form. The section header is [menu|*formname*], where *formname* is the name portion (including any ~*value* suffix) of the name,title assignment in the selection lines of [menu]. Within each section, the following *name*=*value* pairs can be used:

- desc=*description* - provides panel descriptive text.
- title=*title* - overrides the default title from the [menu] selection line

- icon=*file* - names an icon file for the panel. Standard icons can be found in web/en-us/icons. A full path is not necessary if the file is in that or another standard UnForm directory, such as the install location and the web/en-us paths.
- newwin=0|1- if set to 1, the panel opens a new window for the form.
- lb=*library* - names the library the form is associated with (this value is required).
- doctype=*doctype* - names the document type the form is associated with.
- All other *name*=*value* pairs are added to the URL that is used to run the form. Values can be retrieved from the form documents search property using JavaScript, or more conveniently, if the form loads the common.js script library, use the queryValue(*name*) function.

The following menu section describes the menu panels that will be presented to various users and groups. For example, the group AP will see two panels, My Docs Status Report and Capture to ERP Batch Processing, which will run the mdf7-v2 and mdf8-v2 forms, respectively.

A webform name of "break[,*title*]" displays a line break and title bar, rather than a menu panel.

```
[menu]
group:AP=break,AP Workflow
group:AP=mdf7-v2,My Docs Status Report
group:AP=mdf8-v2,Capture to ERP Batch Processing
user:admin=mdf9-v2,All Users Doc Status Report
user:collins=mdf9-v2,All Users Doc Status Report
user:mje=mdf9-v2,All Users Doc Status Report
user:heyman=mdf9-v2,All Users Doc Status Report
```

The configuration for the mdf9-v2 form is shown below. The panel title will be the default, All Users Doc Status Report. The panel description and icon are shown. The web form will use library AP-MDF and will be shown in a new window. When the form is displayed, two additional URL fields will be available, Runtype=0 and Sortype=1, which code in mdf9-v2.html can reference.

```
[menu|mdf9-v2]
desc=Status report for my AP documents
icon=Play_24.gif
lb=AP-MDF
newwin=1
RunType=0
SortType=1
```

MESSAGE TRANSLATIONS

UnForm 8.0 supports the ability to use user-selected message files for language translations of user interfaces in the following tools: Windows Server Manager, Windows Support Server, Design Tool, and Image Manager.

Each of these programs has a configuration .ini file that can contain a [languages] section. This section contains lines in the format of *ext*=*title*, where *ext* is a file extension to use when opening the messages file. UnForm is always supplied with English messages files with an "eng" extension.

The interface for each program offers a drop-down box to select a current language from those configured. The selected language is saved for future executions. The following table shows the configuration files that can contain the [languages] section, and the base names of the language files.

Program	**Configuration File**	**Language Files**
Windows Server Manager	uf80dx.ini	uf80dl.*
Windows Support Server	uf80ss.ini	ufssmsg.*
Design Tool	ufdsn.ini	ufdsnmsg.*
Image Manager	uf80scn.ini	ufscnmsg.*

There is also a Windows client messages file, uf80cmsg.*. The default file is uf80cmsg.eng, but there is a uf80c.exe command line option to specify an extension (-lang *ext*).

DYNAMIC RULE FILE TRANSLATIONS

At runtime, a translation file can be associated with a rule file in order to perform dynamic substitution of text and barcode values, and also anchor text to which many enhancements can be related. The purpose of this feature, added in version 8.0.25, is to ease the effort of translating rule sets for different languages. By editing a messages file that is associated with hard code text fragments in a rule file, UnForm can perform word and phrase substitutions dynamically.

The structure of the translation file is an "ini" file, with sections that match the names of rule sets in the current rule file. At runtime, UnForm will load assignment lines from the section that matches the current rule set name, plus any assignment lines found in the file before the first section header. For example, if a rule file contains the rule set "Invoice", then lines in the [Invoice] section of the translation file are used. Any assignment lines at the top of the file are appended to these lines, allowing for global settings to be used as well as ruleset-specfic settings. An example of a translation file is the samples/advanced.spanish.ini file, which contains translation samples associated with some rule sets in the advanced.rul file.

A translation file can be specified in the uf80c command line with a –trans "*filename*" option. In addition, the active file can be changed at runtime in any code block, using the settrans("*filename*") code block command.

The format of an assignment line is simply a *name*=*value* pair. The *name* value can include a context prefix, where the context can be one of the words text, barcode, or anchor, followed by a tilde (~). For example, the line text~Invoice=Factura would replace "Invoice" in a text command with "Factura". A line without the context prefix (just Invoice=Factura) may apply to both text and barcode commands.

Values are case sensitive, and only whole values are replaced. For example, if a text command specifies "Invoice No:", then the assignment line might be: Invoice No:=Factura no:.

When assignment lines are evaluated, they are first evaluated in a context of text, barcode, or anchor. If a match is found, that assignment is used. If not, text and barcode values are next evaluated without a context, and if a match is found, that assigned value is used. Without a match, the original text is used.

Anchor context substitutions apply to content that is searched on the page and *only* work in context mode (the "anchor~" prefix is required). While text and barcode substitutions apply to content added to the job, anchor substitutions apply to content in the print stream of the job. Anchor substitutions do not change text values, but instead change the text values searched for. Many commands support anchors in their syntax, when the first command is a quoted string. For example, a font command might look like this:

```
        Font "Continued@50,60,80,66",0,0,9,1,cgtimes,12,bold
```

This command will search each page, in columns 50-80 and rows 60-66, for the word "Continued". When it finds the word, it applies a font at 0 columns and 0 rows offset from the position, for 9 columns and 1 row. When a translation file is active, UnForm will look for a line "anchor~Continued=*value*, and search instead for *value*. Additional numerical adjustments are performed on anchor contexts:

- The difference in lengths of the original text and new text is calculated
- The difference is added to the ending column if @col,row,endcol,endrow syntax is used
- If the offset column is 0, the difference is added to the number of columns, to accommodate enhancements designed to apply to the text that is located.
- If the offset column is not 0, the difference is added to the offset column itself, to accommodate a new relative position based on the new text.

Anchors can include the following prefixes: ~, !=, and !~. These prefixes modify how the anchor text and the search are performed, and are not considered part of the assignment name. A ~ prefix indicates the text is a regular expression, and the != and !~ options indicate "not", so match all positions where the search does not find the sought string or regular expression. Since regular expressions can result in variable lengths, the above described numerical adjustments are not performed with regular expression anchors. Another limitation of anchor-based translations is that they are not supported in Application Formatted Output (AFO) jobs, though text and barcode translations are supported.

Since both "~" and "=" characters have special meaning, if they should be part of the actual name they must be escaped with a backslash. For example, if the *name* value should contain an equal sign, which normally separates it from the *value*, that equal sign must be escaped: "Cost=", for example, should appear as Cost\= =*value*.

It should be noted that the dynamic translation feature is a simple substitution function, and it does not account for cases where a substitution dramatically alters the space used by a particular text fragment, or when a different font or character set should be used. In some cases, it might be necessary to use fit or wrap options, expressions for positioning, or code block exec() functions to accommodate some aspects of translation.

Code blocks can also utilize the features of text translation through three functions:

- settrans("*filename*") sets the translation file dynamically as the job runs. This overrides what might be set via a -trans command line option.
- gettrans() returns the active translation file.
- translate(*name$* [,*context$*], *forcecontext*) returns the value associated with the specified name, based on the translation file and current rule set. The context value can be "text", "barcode", or "anchor", and if *forcecontext* is true (non-zero), only context-based names are searched.

There is a rule file, samples/trans.rul, with the rule set "trans", available in the samples directory. This rule set is designed to scan any rule file sent through it as input, and build as output a skeleton translation file based on hardcoded text found in the rule sets. In order to run this rule set, use this command line:

uf80c -i "*rulefile*" -f trans.rul -r trans -o "*inifile*"

TCP/IP MONITOR

UnForm includes a TCP/IP monitor program that can watch for raw print jobs arriving from network computers, similar to how an HP Jet Direct card would. In effect, the UnForm server can serve one or more virtual Jet Direct ports, each with an associated UnForm client command line.

The monitor is automatically started if there are one or more port configuration lines defined in the [tcpports] section of uf80d.ini. For example:

This line would print to the server's spooler –dlaser device, processing jobs through the acme.rul file:

```
	9100=-o ">lp -dlaser -oraw" -f acme.rul
```

This line would print to a Windows server shared UNC printer, processing jobs through the acme.rul file:

```
	9101=-o \\winsrv\laser1 -f acme.rul
```

This line would generate pdf files to the path specified, using the date and job number to generate unique names:

```
	9102=-o "/usr/pdfs/%d.%j.pdf" -p pdf
```

The following substitutions are made in the command line definition:

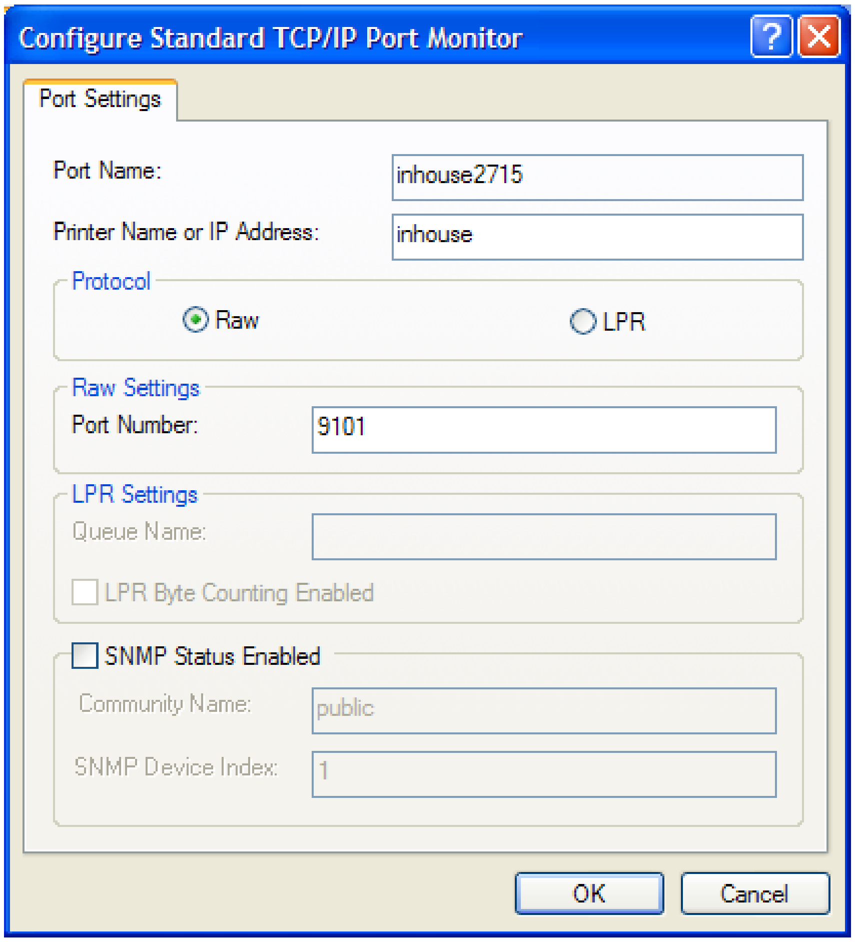
Characters	Substitution
%d	The date in YYYYMMDD format.
%t	The time in HHMMSS format, using a 24 hour clock.
%p	The process ID (this is not necessarily unique).
%j	The sequential job number, which is an ever-increasing unique number.

When jobs are submitted to the UnForm server in this manner, it is important to realize that the submission is one-way, and once printed the job resides entirely on the server. It is therefore not possible to print a job and have data returned to the client (i.e. –o client:*device*), or to have PDF previews generated on the submitting workstation (-p winpvw). Once the job is submitted to the TCP/IP monitor, it becomes local to the UnForm server, as if uf80c is physically run on the server (which, in fact, is what happens).

When jobs are submitted, they are dropped into the rpq/ subdirectory under the UnForm server installation. All submission files are given a unique name with a ".in" extension, and a companion file with a ".cmd" extension is also created that contains the command line options. As jobs are received, and also at least once every 5 seconds, a sweep is made of newly submitted jobs, each submitted to the server via the server's local "uf80c" program. As a byproduct, you can drop jobs into this directory independently of the server, being careful to create the ".cmd" file first, then the associated, complete ".in" file, using your own unique naming algorithm. Note that the sweep assumes that any *.in file is a complete file and will have an associated .cmd file, so it is incorrect to open a .in file and begin writing to it, as the sweep may attempt to process an incomplete file. Instead, create the file with a different extension and then rename it when it is ready for processing.

To configure Windows printers to submit jobs to this monitor, you can use the built-in Windows support for TCP/IP printers. When configuring a printer, you can choose to Add a Port, selecting Standard TCP/IP Port. The Printer name or IP address of the "printer" will be the UnForm server, the Protocol is "Raw", and the Port Number is the number of the configured port line defined in uf80d.ini. UnForm can accept two types of input: plain text and PostScript, so you can choose either the Generic / Text Only print driver, or a PostScript driver, such as the Generic MS Publisher Imagesetter or one of the many printer vendor PostScript drivers. Note there are significant differences in the way UnForm handles the two different types of input. See the UnForm AFO chapter for more information.

The picture below shows a Windows XP example of the configuration screen:



Port Name: inhouse2715

Printer Name or IP Address: inhouse

Protocol: [✓ Raw] LPR

Raw Settings
Port Number: 9101

LPR Settings
Queue Name:
LPR Byte Counting Enabled

SNMP Status Enabled
Community Name: public
SNMP Device Index: 1

Note that other operating systems also support methods of supporting raw TCP/IP printers. For example, Linux contains the "jetdirectprint" script that is used by LPRng to send jobs raw TCP/IP devices.

INTEGRATING UNFORM WITH APPLICATIONS

UnForm is capable of interfacing with any application that can provide it with text input or PostScript. On UNIX, this integration is generally performed via pipes. On Windows, your application can use TCP/IP printing, or can print to a file, and then launch uf80c.exe when the printing is complete.

If your application prints by opening a pipe to the spooler, just insert UnForm into the pipeline:

Before: |lp –dprinter –s 2>/dev/null

After: |uf80c –f *rulefile* | lp –dprinter –oraw –s 2>/dev/null

|uf80c –f *rulefile* –o '>lp –dprinter –oraw'

The second option, above, submits the job for printing on the server, while the first option will wait for the server to return the job for local printing on the client.

If your application prints to a device, such as "/dev/lp0", then you can probably modify it like this:

Before:/dev/lp0

After: |uf80c –f *rulefile* –o /dev/lp0

Note the use of the –oraw option in the above spooler examples. It is important for UnForm's output to be handled as binary data by the spooler. The –oraw option is used by some UNIX spoolers, such as the SCO LaserJet model script, and the CUPS printing system. Other spoolers require different options, such as "-o-dp" for AIX, –T pcl for Unixware, -b for some older Linux installations. Check your lp configuration tools or man pages for the appropriate settings for options such as "binary", "raw", or "pass-thru" printing.

In the case of direct device output, you will need to develop a site-specific mechanism for turning off post-processing on the device, either permanently, or while an UnForm-modified job is printing.

If your application cannot print to a pipe, or runs on Windows, then your application can be modified to print a text file, then execute UnForm when complete. Your environment may provide a way to do this automatically, such as the EXECOFF mode in Visual PRO/5 noted earlier. Here is a simple Visual Basic example of creating a file and launching UnForm:

```
open "work.txt" for output as #1
print #1,tab(35); "INVOICE"
… more printing …
close #1
if shell("uf80c.exe –i work.txt –o //server/hplaser –f  rulefile",6)=0 then
```

```
        end
else
        msgbox "UnForm failed to start."
end if
```

Integrating UnForm with BBx

BBx handles printers via *alias* lines in a configuration file, typically called config.bbx. Printer alias lines identify a name, an output designation, a description, and several mode options. To incorporate UnForm into the configuration file on a UNIX system, you need only include an UnForm command line as part of the output designation.

BBx output designations can specify files, physical devices, or pipes, and UnForm can be installed to work with any type of definition. Note that any escape sequences configured in modes like PTON, SP, and CP are sent to UnForm and therefore need to be PCL sequences. UnForm understands how to strip a job of PCL codes, but not other printer codes. In some cases, when UnForm sends a job straight through without enhancements, these PCL sequences will also be passed on.

UNIX Aliases

A printer alias line on UNIX generally pipes to a program, such as the uf80c client program. This client program in turn can pipe its output to the spooler, or to a file, or it can instruct the server to handle the output from its end, by specifying the –o option.

Here is a sample alias line that pipes through UnForm to the local spooler:

alias P1 "|uf80c -f my.rul | lp -dxyz –oraw -s 2>/dev/null" "Printer Name" … ***various modes …***

Here is a sample alias line that instructs the server to print the job to its spooler. The advantage of this type of configuration is that the client doesn't have to wait for the job to finish. It submits the job to the server and exits quickly.

alias P1 "|uf80c -f my.rul -o \'>lp -dxyz –oraw\'" "Printer Name" … ***various modes …***

Note the use of the –oraw option in the above examples. It is important for UnForm's output to be handled as binary data by the spooler. The –oraw option is used by some UNIX spoolers, such as the SCO LaserJet model script, and the CUPS printing system. Other spoolers require different options, such as "-o-dp" for AIX, –T pcl for Unixware, -b for some older Linux installations. Check your lp configuration tools or man pages for the appropriate settings for options such as "binary", "raw", or "pass-thru" printing.

UnForm can also print directly to a device, as in this example:

alias P1 "|uf80c -f my.rul -o /dev/lp0" "Printer Name" … ***various modes …***

Note that this line will behave differently with the UnForm pipe than without. When opening and sending output directly to a device, printing will occur immediately, without closing the device. However, with the pipe to UnForm, the output will not appear until the device is closed. The application may need to be modified to account for this if UnForm is to be used in this circumstance.

Windows Alias Lines

Under Windows, where pipes are not available, change the printer definition to create a file, and then use a post-processing mode, called EXECOFF, to execute UnForm with options to read the file and output to a device.

A Windows alias line will look similar to this:

alias P1 C:/TEMP/P1.TXT "UnForm Printer" CR, LOCK=C:/TEMP/P1.LCK, O_CREATE, SPCOLS=132, SP=1B451B287331362E3636481B266B3247, EXECOFF="uf80c.exe -ix C:/TEMP/P1.TXT -o *device* -f my.rul"

In the above example, a file called P1.TXT is created, using the mode O_CREATE to create the file if it doesn't exist, and using a lock file to prevent two users from writing to the same file at the same time. Note that if a file is specified with a local workstation path, such as C:\\P1.TXT, then a lock file is probably unnecessary. Just remember to specify the same path in the –ix option. Once the printer is closed by the application, the code specified by the EXECOFF mode is executed, which runs UnForm as an executable, using the P1.TXT file as input and the printer as output.

Note that pathnames containing backslashes will need double backslashes, due to the way BBx parses the command line. For example, to refer to "uf80c.exe -i c:\data\p1.txt ...", you would need to specify "uf80c.exe -i c:\\data\\p1.txt ...". You can also use forward slashes in place of backslashes, and you don't need to double them.

The *device* in the –o argument can be one of two things:

- An LPT*n* port, which can be mapped to a UNC device name with the Windows "net use" command.

- A UNC device name, defined by sharing a printer, so the name becomes //*system*/*printer*, where *system* is the system with the shared printer, and *printer* is the "share name" of that printer.

Another variety of alias line can generate a temporary PDF file and display it on the client PC, assuming you have an Adobe Acrobat Reader installed. This alias doesn't require a –o argument, but will honor it as the client-side file name for the PDF document generated. The driver selected by the –p option must be either win or winpvw, like this:

alias PUNF C:/TEMP/PUNF.TXT "UnForm Printer" CR,LOCK=C:/TEMP/PUNF.LCK,O_CREATE,SPCOLS=132, EXECOFF="uf80c.exe -ix C:/TEMP/PUNF.TXT -p winpvw -f my.rul"

Note that **the uf80c client software must be installed locally on any workstation that will execute it** to submit jobs.

Integrating UnForm with ProvideX

Simple UNIX Integration
On UNIX systems, you can integrate UnForm within the link file as the output device, and use a standard LaserJet or plain text print driver. The device used in the link file would be simply a re-direct to the uf80c program (if using ProvideX 6.0 features, a pipe (| rather than >) can be used as well), such as ">uf80c –f acme.rul –o '>lp –dhp –oraw'".

Note that this option was not available in prior versions of UnForm.

Integration using the ProvideX Print Driver uf8ptr
This method works for both UNIX and Windows environments, and provides more program control over the UnForm options when executing the uf80c client.

UnForm installation includes a ProvideX print driver **uf8ptr** which should be copied to your ProvideX lib/_dev directory. This driver provides platform-independent support for UnForm, along with additional capabilities for managing UnForm command lines from the ProvideX application. In addition, it supports WindX-based output. Once copied to your ProvideX lib/_dev directory, this driver is available to use when defining ProvideX link files, which are used as printers in ProvideX.

To use the uf8ptr print driver:

When a link file is defined, you specify an output device and a driver program. The output device is generally something system specific, like ">lp –dhp –oraw" on UNIX, or //SERVER/PTR on Windows, or it can be a special driver name for Windows, such as *windev**, or [WDX]*windev*. In some cases, it can be /dev/null or NUL, if the driver will be directing output somewhere for the user.

The uf8ptr driver determines a default output device based upon the link file's specified output, and then re-routes the printer output to a temporary work file.

It then looks for a configuration file for additional uf80c command line parameters. This file is simply a text file named *linkfile*.unf. For example, for a link file named P1, uf8ptr will look for a file called P1.unf for additional parameters. In this text file can be one or more lines with uf80c command line options.

Once the file-based parameters have been loaded, uf8ptr then looks for the OPT value that was used in the OPEN directive, if any, for additional parameters. Any parameters named in the OPT value will override those found in the configuration file.

When all parameters have been resolved, a uf80c command line is built for execution at the end of the job. In cases where the output needs to be returned to a WindX client, the driver handles uf80c appropriately to create local output and copy that output back to the WindX PC.

Example 1:

LP is a link file pointing to device /dev/null.
LP.unf contains: -p pdf.

```
invoiceno$="00015"
OPEN(1,opt="-o /archive/"+invoiceno$+".pdf")"LP"
```

The result will be an uf80c command like this, which executes when the printer is closed:

uf80c –i *workfile* –p pdf –o /archive/00015.pdf

Example 2:

P1 is a link file pointing to device ">lp –dhp4000 –oraw".
No P1.unf file is defined
OPEN(1,OPT="-f acme.rul")"P1"

This will override the default rule file defined at the server, using acme.rul. Output will go to ">lp –dhp4000 -oraw" on the machine where the UnForm server is running. Typically this is the same machine that runs ProvideX. If it is not, add a –server *servername* option to OPT or *linkfile*.unf. In such a case, if the >lp command isn't valid locally, you will need to add a –o option to the configuration and change the link file to point to /dev/null (or NUL on Windows).

Example 3:
P2 is a link file pointing to device [WDX]*windev*.
Opening P2 will result in laser output being produced and sent to the WindX printer selected.

Example 4:
P3 is a link file pointing to device NUL.
P3.unf contains –p winpvw.
Opening P3 will cause production of a temporary PDF file. This file will automatically be viewed on a WindX client or in a Windows ProvideX session.

LICENSING

UnForm is licensed based on the number of concurrent jobs it can process, with counts available as 1, 3, 5, 10, 15, 20, 25, 30, 40, 50, 75, 100, and unlimited. The UnForm Design Environment checks out a special "Designer" license, and it is available in different concurrent counts as well.

Licensing is controlled entirely by the server process, uf80d. You can install the uf80c client programs freely anywhere on your network.

Each UnForm installation has a serial number. There is one special serial number, UF0099999, reserved for demo mode use on any machine. All permanent licenses are assigned a unique serial number and must be licensed to a single machine installation. Serial numbers and their associated PIN codes are assigned by SDSI when UnForm is purchased. In order to obtain permanent or emergency temporary activation keys, the serial number and PIN code are required.

There are up to three activation keys that must be entered for full operation of UnForm: a system key, a job key, and a designer key. The system key enables UnForm to operate on a specific computer. The job and designer keys determine the number of concurrent job and design tasks that may run. For demo mode operation, just a temporary system key is required; demo mode operation automatically enables 3 jobs and 3 designers.

There are three types of system activation keys:

30-Day Demo
This license has a fixed serial number (UF0099999) and can run on any machine for 30 days. While running under this serial number, UnForm will print "Demonstration Version" phrases on any enhanced output, and will print a trailer page for each job. This is the first mode activated after an installation, as it enables the retrieval of a System ID and Machine Class needed for permanent licensing later, as well as allowing UnForm to operate in demo mode.

Permanent
This license has an assigned serial number, and requires a System ID and Machine Class to activate. A permanent license does not expire, enabling UnForm to run perpetually on the machine where installed and licensed. The System ID is derived from a given installation machine and attributes of a file in the UnForm rt\lib\keys directory (Windows) or the rt\lib directory (UNIX), so it will change if the installation is moved to a new machine, or even to a new location on the same machine. Once the System ID changes, the permanent activation key will no longer work, and UnForm must be re-activated.

If the original permanent installation of UnForm is no longer used, then you can request a reset of the permanent license to enable a new System ID and Machine Class to be associated with the permanent activation key. Contact sales@synergetic-data.com to request resets.

Emergency Temporary
This license is assigned a serial number, like a permanent license, but it does not require a System ID or

Machine Class to activate. This allows you to re-install UnForm on a different machine than originally licensed, and operate it for 30 days. Once a temporary license has been issued for a given serial number, another temporary license cannot be issued for 45 days.

UNIX Licensing

To activate UnForm on UNIX, perform the following steps:

- Login as root.
- **cd** to the UnForm directory (i.e. cd /usr/lib/sdsi/uf80).
- Execute **./license.sh**.

The license.sh script prompts for the following options:

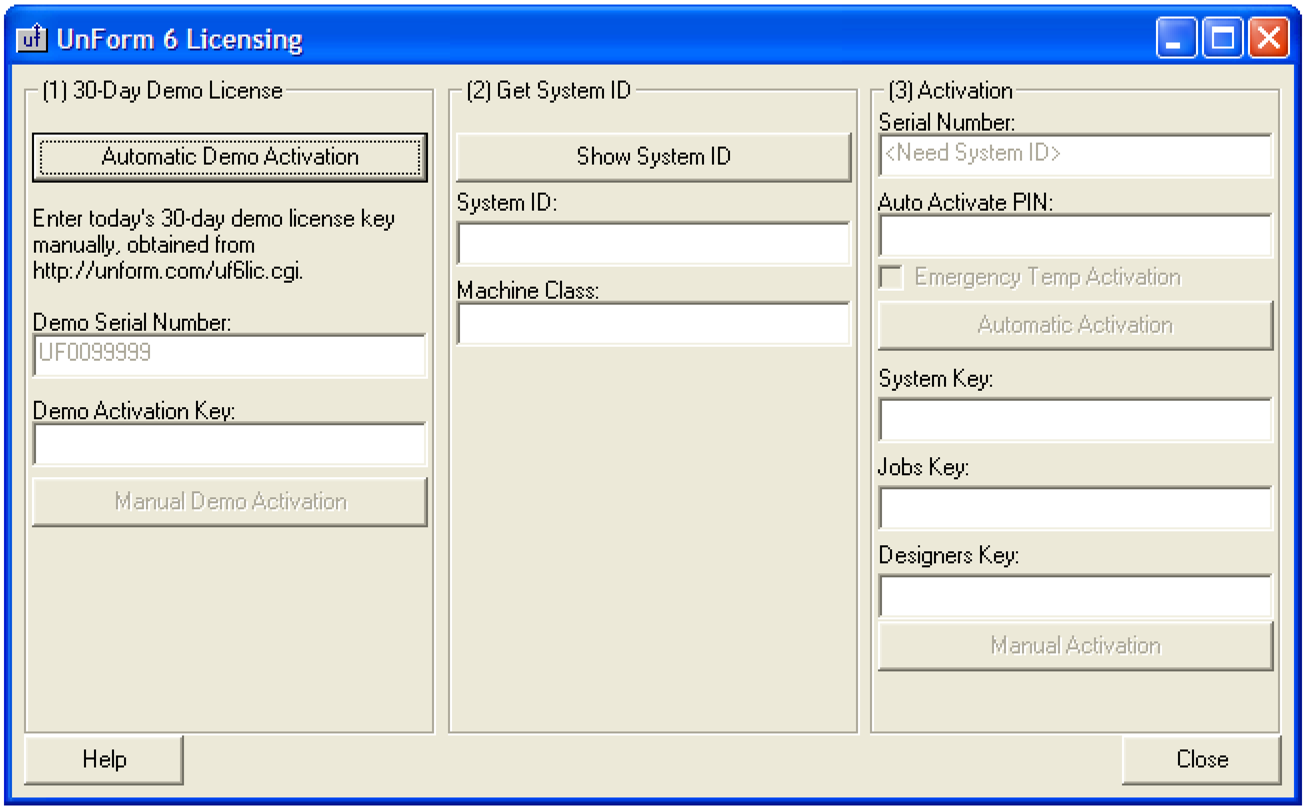
```
UNFORM LICENSING OPTIONS

Use the following options if this machine is connected to the Internet:
-----------------------------------------------------------------------
1 - Permanent Activation (requires serial number and PIN code)
2 - Emergency Temporary Activation (also requires SN and PIN)
3 - 30-Day Demo Mode Activation

Use the following options for manual activation.  Activation keys
can be obtained from http://unform.com/uf8lic.cgi.
-----------------------------------------------------------------
4 - Display System ID and machine class (needed for option 5)
5 - Enter Permanent Activation
6 - Enter Emergency Temporary Activation
7 - Enter 30-Day Demo Mode Activation

q - quit
Enter selection:
```

To obtain either a permanent or emergency temporary activation, you will need to know your serial number and PIN code previously assigned by SDSI. These values are not necessary to obtain 30-day demo mode activation.

If your machine has Internet access, you can perform activation easily by choosing options 1 through 3. Options 1 and 2 will prompt you for your serial number and PIN. Each of the three options will use the Internet to retrieve the desired activation key.

If the Internet is not available from the install machine, then you can perform activation manually by using another machine to visit http://unform.com/uf8lic.cgi. Use option 4 to display the System ID and Machine Class, which will be required to obtain a permanent activation key from this web site. Options 5 and 6 will prompt for a serial number, system key, jobs key, and designers key, in sequence. Option 7 will only prompt for a system key.

Windows Licensing


UnForm 6 Licensing

(1) 30-Day Demo License

Automatic Demo Activation

Enter today's 30-day demo license key manually, obtained from http://unform.com/uf6lic.cgi.

Demo Serial Number: UF0099999

Demo Activation Key:

Manual Demo Activation

(2) Get System ID

Show System ID

System ID:

Machine Class:

(3) Activation

Serial Number: <Need System ID>

Auto Activate PIN:

Emergency Temp Activation

Automatic Activation

System Key:

Jobs Key:

Designers Key:

Manual Activation

Help

Close

The first step after an installation is to activate demo mode. This initializes the system ID file, enabling a permanent license to be obtained. If you get an error message after pressing the Show System ID button, then this installation has never been initialized, and you must activate demo mode first.

To activate demo mode:
If you are connected to the Internet, press the Automatic Demo Activation button. This will obtain a current demo mode activation key from SDSI's website and activate the run-time engine.

If you are not connected to the Internet, go to a computer that is, and go to http://unform.com/uf8lic.cgi, then click the link to get a 30-day trial. Note the activation key returned, and enter it exactly the same way in the Demo Activation Key field, then click the Manual Demo Activation button.

To verify the activation, click the Show System ID button. If the System ID and Class fields get filled in, then it worked.

To activate permanent mode:
To activate automatically over the Internet, you need to click the Show System ID button to get the System ID and Machine Class fields. Then fill in your serial number and PIN code, and click the Automatic Activation button. This will use your information to obtain a permanent activation key for the system, as well as your job and designer activation keys, and activate everything.

To activate UnForm manually, note your System ID and Machine Class, then go to http://unform.com/uf8lic.cgi. Enter your serial number and PIN code, then click the button to get a permanent license. When prompted, enter the System ID and Machine Class exactly as noted on this screen. Note the three activation keys returned, and enter them exactly as provided in the three entry fields, then click the Manual Activation button.

To activate in emergency temporary mode:
To obtain a temporary activation over the Internet or manually, follow the steps for a permanent license, but check the Emergency Temp Activation option. The System ID and Machine Class are not used for temporary activations.

Activation Errors

Permanent activation keys are dependent on the system ID and machine class information generated by an installation. Therefore, a permanent activation key will only work on the original installation for which it was generated. If UnForm needs to be moved or re-installed, a new permanent activation key must be generated. This is only possible if SDSI resets the permanent key for your serial number, so you must contact SDSI, certify that the original installation is no longer in use, and request a reset.

In the meantime, you can obtain an emergency temporary activation to allow your serial number to be used on a new installation for 30 days.

If you attempt to get a new permanent activation key and are notified that one has already been assigned, then contact SDSI to request a reset. If this cannot be done in a timely fashion, get an emergency temporary key instead, and then contact SDSI at a later time.

Note that temporary keys are issued at most once every 45 days. If you get an error message indicating the temporary key availability has not expired, then you must contact SDSI to get a reset.

Additional Evaluation Period
In addition to the 30-day UnForm demonstration mode, if you license UnForm for jobs but not designers, scanning, or archiving, an additional 30-day demo period is enabled for those products. This demonstration period begins the first time one of these components connects to a licensed UnForm server.

UNFORM COMMAND LINE OPTIONS

The uf80d server program can be started with the following options:

UNIX command lines	
uf80d start	Starts the server daemon.
uf80d stop	Stops the server daemon.
uf80d restart	Stops, then starts the server daemon.
Windows command lines	
uf80d.exe -configure	Displays the configuration window for the server. This option is available in the Windows Start menu and the Server Manager.
uf80d.exe	Displays the Server Manager window.
uf80d.exe -v	Executes a local UnForm client (uf80c.exe) to show the server version.
uf80d.exe -start	Manually starts the server if it is NOT installed as a service.
uf80d.exe -stop	Manually stops the server if it is NOT installed as a service.
uf80d.exe -installservice	Manually install as a service
uf80d.exe -uninstallservice	Manually uninstall the service

The uf80c client program offers many options, which control various aspects of how it communicates with the server and how the server is told to execute the job. Note that if the command line becomes too long for the operating system, you can use the –z or –zx options, which cause command line options to be read from a text file.

Standard Options	
Option	**Description**
-300	Causes UnForm to suppress 300 dpi settings within the PCL output file. Some PCL devices don't support the PCL unit of measure command, and instead include it as printed output. If this option is used, any images (dump files) or attachments must also be generated for 300 dpi and suppress any unit of measure settings.
-about	A Windows-only option that displays information about the client, including the location of the active uf80c.ini file.
-c *copies*	Causes UnForm to issue multiple copies of the entire report. This differs from the -pc option. If *copies* is set to less than 2, this option is ignored. This option and the "-pc" option are mutually exclusive; also, rule sets can specify copy options that will override command line options.
-ce *copies-enabled*	Performs implicit skips of any rule set copy NOT found in the *copies-enabled* list. For example -ce “1,2” would force copies other than 1 and 2 to be suppressed. This option is useful in sub-jobs executed with the jobexec() function or via the archive command to suppress certain copies.

-ci or -color	Forces pcl image conversions to retain color rather than force black and white. See the image command for more information about automated image conversion and scaling. This also implicitly sets the –gw option.
-cols *n*	Sets the default columns per page when a job is using default scaling, as when the –p pdf or –p laser options are used and no rule set is detected or specified. See also the –rows option.
–compress or -cmp -nocompress	The –compress or –cmp options will force compression of PDF files, even if the best compression available is RLE. If the operating system supports zlib compression, then Flate compression is turned on by default and this option is redundant. If you want to disable the automatic Flate compression, use the –nocompress option.
-config	A Windows-only option that displays a configuration form and updates the client's uf80c.ini file. See the -about option for the location of this file.
-cover "ruleset [,rulefile [,args]]"	Generates a cover page for the job using the rule set specified. If a rule file is specified, the rule set is read from that rule file. If arguments are specific, the subjob used to generate the cover page will include the specified command line arguments. The arguments can be used to pass parameters and other processing options, but should not include -f, -r, or -p arguments.
-cr 0|1|2|3	Sets handling for embedded carriage returns (chr(13)) in lines read from the input stream. The default value is defined in uf80d.ini, in the [defaults] section, cr=*n* entry. • 0 will truncate lines at the first CR. • 1 will strip CR character, so the line continues as if the character were not present. • 2 will fold lines, and non-space characters are placed in the line buffer, simulating an overstrike. • 3 will fold lines and insert an extra space, which accommodates Windows Generic/Text Only printers that overstrike conflicting characters. Note if UnForm detects line-terminators are CR characters rather than LF or CRLF sequences, this option is not operational.
-debug	Causes job-based submission *serverpath*/temp/ files *.in, *.out, and *.err to be retained rather than deleted after job completion. It also causes generation of the following job-based files: *.eml log file for email operations, *.gs.log for ghostscript operation error and standard output (Unix or console version Windows only).
-e *error-file*	Causes UnForm to output any errors to the file specified. Error files reside on the client system, not the server.

-emattach "*value*" -embcc "*value*" -emcc "*value*" -emfrom "*value*" -emlogin "*value*" -emmsgtxt "*value*" -emoh "*value*" -empswd "*value*" -emsubject "*value*" -emto "*value*" -emlogfile "*value*"	These options supply values for an automatic email command. See the email command documentation for descriptions of each option. The –emto option is required, all others are optional, though certainly the –emsubject and -emmsgtxt are likely required for a given application. For emailing to work, the job must be a PDF job, and the server's mailcall.ini file must be properly configured with a server= line defining the SMTP server.
-f *rule-file* -f "*file1*;*file2*;…"	Establishes a different rule file than the default specified during the installation. Rule files are text files that contain descriptions of the form enhancements for one or more forms. The enhancement options are described in detail under Rule Files, below. UnForm will always search for the rule file first in the UnForm server directory, then by the full pathname given. Rule files must reside on the server machine, not the client. If multiple rule files are specified, delimited by a semicolon, they are merged, with the global regions merged in reverse sequence. This feature was added at 8.0.24. By convention, rule files have a .rul suffix, though this is not a requirement, and the *rule-file* value can be any file name. The UnForm Designer tool maintains a .rud suffix for working rule files and a .rul suffix for published rule files.
-gb|-greenbar [*options*]	Adds alternating shade patterns to simulate green bar paper. If the *options* parameter is supplied, it should be in the form defined by the shade command for repeating shade values. If no option value is supplied, the default is 3 lines shaded at 10%, 3 lines skipped, repeated until the end of the page.
-gs	Causes UnForm to generate laser driver shade regions graphically, rather than using internal PCL shade commands. The result is finer shading detail, especially at 600 dpi. Using this option will add between 2K and 4K per job. The gs command can also be used in rule sets to control graphical shading at a copy level.
-gw	Forces UnForm to pass through PCL image width and height escape sequences to the printer. This is generally necessary on color laser images to avoid a black stripe from the right image edge to the right margin. However, if you are using PCL images, then it is important that all images on a form contain width and height values so they won't conflict with one another. Some image generating programs don't store the width and height values.

-i *input-file*	Names an input text file for UnForm to process as input. If not specified, or if it is a dash (-i -), then standard input (std input) is read. Under Windows, standard input cannot be used, so an input file must be supplied. Note that the input file must reside on the client's computer, not the server.
-ix *input-file*	Same as the –i option except the input text file is removed upon completion of task. Note that the input file must reside on the client's computer, not the server.
-land	Turns on landscape print mode as the default. A portrait command in a rule set will override this option. Note that landscape printing usually requires a reduction in the number of rows per page, as compared with portrait printing, in order to produce usable results.
-lang *languageext*	Used by the Windows client, uf80c.exe, to render messages and window titles from the file uf80cmsg.*ext*. The file uf80cmsg.eng is always supplied with UnForm, and the default language is English. Use the option first in a command line in order to set the language file correctly for any messages related to later options.
-lib "*dir*[;*dir*;…]"	Add directory (or directories delimited by semicolons) to the library of search paths used for locating external files, such as images, attachments, or merge rule files. Note that the library= line of uf80d.ini provides another method of doing this, and the rule file's path is also automatically added to this search list.
-lockcols	When the label dimensions and cols setting are established, UnForm scans mono-spaced internal fonts for the closest match that will not exceed the cols specified, then recalculates the cols to agree with the font selected. This allows print stream text and all other enhancements to scale together. However, it also causes labels to shrink in printable area width, sometimes very noticeably, resulting in graphical commands not being placed where expected. This option was added to prevent this recalculation from occurring, at the expense of losing the print stream scale matching. With this option, graphical commands will print where expected on the label, but may not align with print stream output. Zebra-only command.
-m	Sets the printer model to a name that can be found in the ppd directory (without the ".ppd" suffix, e.g. –m hp4000 will load the ppd/hp4000.ppd file). This is useful when producing PostScript jobs that use printer features such as duplex or tray selection, as the code for those features is defined in PPD files provided by printer manufacturers. If no –m is provided, then UnForm will select a default PPD file based on the driver. Custom PPD files can be obtained from a printer vendor or from various Internet sources, or can be written from scratch or based upon one of the generic files.
-macros	Turns on macros.
-macrocopy *n*	Used in conjunction with the –makemacro option. A macro will be created for the designated rule set copy.

-makemacro *n*	Causes UnForm to simply create the appropriate macro for the designated rule set and designate it as the number *n*. It must be used jointly with the –r option and can be used in conjunction with the -macrocopy option. See special section discussing macros later in this documentation.
-nn	Indicates that an error message should be issued if the input stream is empty. The value used for the error message is in the [defaults] section of ufparam.txt, in the entry nullmsg=*message text*.
-noafo	Suppresses the automatic assumption that Postscript input should initiate an AFO job. This flag can also be useful when using jobexec() to generate non-AFO subjobs when run from an AFO job.
-noarc	Turns off any archive commands for the job.
-nocompress	See the –compress option.
-nohpgl	Reverts to full PCL, rather than a mixture of PCL and HP/GL output. A number of laser printed features use HP/GL, which is a standard feature of the PCL5 language. Some PCL interpreters, such as those that may be included in some fax or viewing software, may not support HP/GL, so this option can be used to force standard PCL5 coding for many options, such as box drawing and text alignment. A few features, such as rounded corner boxes, require HP/GL and are not supported if this option is specified.
-nointr	With this flag set, the Unix/Linux client will ignore interrupt signals once the connection to the server is established. This allows it to keep running even if a parent process receives an interrupt signal on platforms that propogate the signal to child tasks.
-notextjob	With this flag set, UnForm will not construct the textjob$[] array, saving time during parsing of the input stream.

-o *output-file*	Specifies an output file or device. If not specified, then standard output (stdout) is used. Under Windows, an output file must be supplied unless one of the special drivers, win or winpvw, is used. On UNIX, the output can be a redirect or pipe to another program, such as lp or lpr. The output device can be specified in the form "tcp:*nameorIP*:*port*" to direct output to a network printer at the name or address specified. If the optional :*port* is not supplied, port 9100 is used, which is the default network printing port. Example: -o "tcp:192.168.1.45". Output names that contain spaces or characters that are meaningful to the operating system must be quoted. The output file or device will, by default, be opened on the server machine. If the name is prefixed with the phrase "client:", then it is returned to the client for local handling. Here are some examples: Server output: **-o ">lp –dhplaser –oraw –s 2>/dev/null"** **-o "/tmp/archive/12345.pdf"** Client output: **-o client://prntsrv/laser** **-o client:c:\archive\12345.pdf** Note that if a Unix pipe or redirect (| or >) is not quoted, output is actually to the client's standard output handle, so it is implicitly client-based. To print to a server-based spooler or program, be sure to quote the argument, as shown above. Server-based output can contain characters that will be substituted at run-time. These character replacements are: • %d for the date in YYYYMMDD format • %t for the time in HHMMSS (24-hour clock) format • %p for the process ID of the UnForm task generating the file • %j for the UnForm job number, a sequential counter Note that on UNIX, if there is no –o specified, or if the output is simply a dash (-o -), then output goes to the client's standard out. A special output of /dev/tty is also recognized as client-side output to the /dev/tty device, often used for slave printing (see the –slon/-sloff options). If the output will be handled by the server, the client will generally exit as soon as the job has been successfully started on the server. If the output is to be returned to the client (or the –wait option is specified), then the client will wait for the server to finish.

-o "*windev*;*name*" -o "*winprt*;*name*"	When UnForm is installed on a Windows system, two special print devices are available: *windev* and *winprt*. Note that the * characters are optional in version 8.0, and either a semicolon or colon can be used. Windows print queues can be referenced in raw mode (*windev*) or via a Windows print driver (*winprt*). When in raw mode, a PCL5 or PostScript driver must be used, specified with the –p option (some printers support direct printing of PDF output as well). When using a Windows print driver, Ghostscript must be installed and configured, which allows UnForm to create temporary PDF output, convert that to image output, and print the images using any Windows print driver (the –p option is ignored). This Ghostscript-driven method is suitable for local printers, but not for remote printers accessed over a slow network connection, due to the size of output generated. Each page is a raster bitmap, and such full page images are large files (i.e. letter-size, 300 dpi, black and white is about 1MB, color about 8MB). The *name* value specified must match a Windows printer name (not a share name). If not specified, the default printer for the UnForm server process will be used (not that of the submitting user). No selection window can be presented for dynamic selection, as UnForm jobs run in background on a server. When using *winprt*, limited printer control is offered on a job-wide basis, for tray, duplex, and orientation. Color output is generated if a –color command line option is used. As the output is produced initially in PDF format, it is not possible to change the output device in mid-job. You can set output dynamically at the start of a job in order to override the –o command line argument, using an output command or setting output$ in a prejob code block, using the same "*winprt*;*name*" syntax. Tray numbers differ from PCL trays, often being manufacturer-defined values above 256. A list of tray numbers for a given Windows printer can be obtained using the system object's winprttrays$(printer$) method.
-o unc:\\server\share	This special Windows UNC printer syntax is designed to print indirectly to the device via a work file and a Windows copy command. This technique works around a limitation of Windows 2008 that prevents printing to UNC printer shares. If UnForm detects it is running on 2008 or higher, it automatically implements this technique, but if needed, the unc: prefix can be specified to force this processing method.

-p *output-format*	Specifies the output format for the job. It may be one of the following values: **laser** (or **pcl**), which produces PCL5 or PCL5c (color) output. The default format is PCL5, but if this option is specified, and no rule set is detected or specified, then the output is scaled to fit the page in conjunction with the –cols and -rows options, or the content itself. Without any –p option, and without a rule set, the job is passed through unmodified. **pdf,** which generates files viewable by Adobe Acrobat Reader or PDF viewers. If no rule set is detected or specified, then a scaled text job is created, based on the –cols and –rows options, or the content itself. **ps** or **ps3**, which generates PostScript output. If no rule set is detected or specified, then a scaled text job in PostScript format is created, based on the –cols and –rows options, or the content itself. The ps3 version leverages the zlib compression support of PostScript level 3, which is supported on many printers, for monochrome images. **eps**, which generates a PostScript EPS image from the first page of output. **zebra*n***, which produces ZPL II output at *n* dots per mm (6, 8, or 12 – default of 12) for Zebra label printers. For special Zebra media handling, you can append the following to **zebra*n***: • Media tracking (Y=standard, N=non-standard label stock). Standard label stock is non-continuous, meaning the media has some type of physical characteristic (web, notch, perforation, mark, etc.) to separate the labels. NOTE: changing between standard and non-standard requires recalibrating the printer. • Set print modes (T=tear-off, R=rewind, P=peel-off, C=cutter). The default values are YT. For continuous labels, 8 dpmm, with a cutter, you would specify **–p zebra8NC**. **html,** which generates Web pages from reports, based on a special set of rule set keywords.

-p *output-format* (continued)	**win, winpvw**, which automatically produces a PDF file and launches the Acrobat PDF viewer on the Windows client. **win** (note win5 is a synonym for win) will print the document using the Acrobat /p command line option. This generally provides a printer selection dialog before printing. On some versions of Acrobat a window is left open after printing. A second format, "**win:*printer***", uses the Acrobat /t option to print directly to the printer named *printer*. Printer names generally match the names in the Windows printer select dialogs, but sometimes can be UNC names. To print to the default printer, specify **win:default** or **win:dflt**. **winpvw** will provide a print preview. These options only work in Windows clients. The **winpvw** option has special significance when retrieving documents or lists from an UnForm archive, via the -arcget, -arcsearch -arclist, or -arclistdocs options. In this case, the preview is generated based on the type of data returned, using the file associations of the Windows shell for all data except PDF documents, which are viewed using the standard UnForm client viewer. For list formats, pipe and tab are viewed as pure text, and csv, html, and xml are viewed using the Windows shell association. Special Ghostscript-driven drivers are also available if Ghostscript is available on the server machine, and if you have configured the uf80d.ini file [drivers] section. The configuration specifies the path to Ghostscript and a set of driver names with Ghostscript -sDEVICE names, a multi-page flag, and a resolution. For example: [drivers] gs=gs bmp=bmp256,1,300 If the command line contains **–p bmp –o imagefile.bmp**, then UnForm will generate an interim PDF file, and execute the gs command to convert that to the format bmp256, with output files imagefile-1.bmp, imagefile-2.bmp, and so on. The images will be produced at 300 dpi resolution. Many standard drivers are configured, and you can add more as needed and as supported by Ghostscript.
-page *lines*	Specifies the number of lines per page that UnForm should read from the input. Normally, UnForm will find form-feed characters to delimit pages. However, if the application simply prints even numbers of lines per page, this can be used to define that value so UnForm can properly parse the input stream. The rule file page command is normally used rather than this command line option, since different reports can have different page sizes. However, this option is useful when doing cross hair prints (the -x option) to properly parse individual pages.

-paper *paper* -ps *paper*	Specifies the paper size used by the printer. Valid values include letter, legal, ledger, executive, a3, and a4. The default is letter. For a complete list, see the [paper] section of ufparam.txt. The paper size can also be specified with a *widthxheight* setting, with an optional suffix of "cm" or "mm" to specify the value in centimeters or millimeters (i.e. 20x30cm).
-pc *copies*	Causes UnForm to issue multiple copies of the report, page by page. If *copies* is less than 2, this option is ignored. This option and the "-c" option are mutually exclusive; also, rule sets can specify copy options that will override command line options.
-pdfauthor "*value*" -pdfkeywords "*value*" -pdfprotect "*value*" -pdfsubject "*value*" -pdftitle "*value*" -pdftrans | -pdfnotrans	These options supply default values for the author, keywords, protect, subject, title, and transparency commands, respectively. All options are used exclusively with PDF output.
-ping	Queries the server, returning a 10-digit string (plus a newline) , made up of 5 digits of total job licenses and 5 digits of job licenses in use at the time the connection is made. The –server and –port options are honored. On Windows, using uf80c.exe, the –o option and –e options are honored.
-port *n*	Specifies the port that the server is listening on, if other than the default of 27280. The –server line can also be used to specify the port, in the format *server*:*port*. The uf80c.ini file also can contain the default port to use in the absence of this option.
-printblanks -pb	Causes UnForm to process blank pages the same as non-blank pages. Normally, blank pages are suppressed.
-prm "*parameters*"	Provides the ability for the application to send parameters to UnForm on the command line. This might be used, for instance, to pass a company number for use in a code block. The format for *parameters* is "*parameter-1=value-1[;parameter-2=value-2;...]*" Any number of parameters can be specified within the limits imposed by the operating system for command line length. Each *parameter* becomes a global string in Business Basic (use the GBL() function to retrieve), and each is set to the *value* specified. Multiple parameters need to be delimited by semi-colons (;). **-prm "company=01;name=Acme Paint"**, for example, would establish two global strings: company and name. These could be referenced within code blocks (prepage, precopy, etc.) as GBL("company") and GBL("name").
-quiet	Forces the Windows version of uf80c to route any errors to the log file defined in uf80c.ini, or "uf80c.log" by default, and to any –e file named on the command line. Without this option, errors are reported in message boxes.

-r *rule-set*	Used to specify a rule set name to use for the job. The rule set specified must exist in the rule file used for the job (see the –f option). If this option is not used, UnForm will attempt to automatically detect what form is being processed based on specifications contained in the rule file. If no form is detected, then UnForm creates a simple text job or may pass the job through to the output unmodified. If the *rule-set* contains spaces, it should be quoted. Rule set names are not case sensitive.
-rd *n*, -rdelay *n*	Introduce a delay of 1 to *n* seconds (a random value) to slow down the pace of jobs submitted to the server when large numbers of jobs are sent by an application. The delay is only imposed if the number of active jobs on the server exceeds a threshold defined by the -rdt option. For example, the following options would implement a random delay of from 1to 10 seconds whenever more than 5 jobs are active on the server: -rd 10 -rdt 5
-rdt *n*, -rthresh *n*	The minimum number of active jobs on the server before the -rd or -rdelay option will be active. If active jobs exceeds this value, a random delay is imposed.
-rland -rport	Turn on reverse landscape or reverse portrait orientation. These options are only valid on laser output.
-rows *n*	Sets the default rows per page when a job is using default scaling, as when the –p pdf or –p laser options are used and no rule set is detected or specified. See also the –cols option.
-s *sub-file*	Specifies a text file to be used as a substitution file. Substitutions are used by UnForm when placing text in the form output. If the text can vary from one form to another, such as company names and addresses, then multiple substitution files can be defined, each containing different names and addresses, and the proper one identified with this command line option. See the **text** keyword for more information. The default substitution file is called "subst". If *sub-file* is not a full path, UnForm will look for it in the UnForm directory. UnForm will automatically generate stbl("@*name*") definitions for each line in the substitution file. Code blocks and expressions can use the stbl() function (gbl() on ProvideX) to return these values.
-serialize	Causes the client to block other client processes until the server has acknowledged the connection, effectively spooling client processes until the server has resources to support additional connections. On Windows, this is managed through a user-specific lock file, so users will not block each other's client executions. Job swarms that can cause performance problems on the server, however, are typically launched by a single process and therefore a single user, so this user-specific locking is effective in those circumstances.
-server *server*	Specifies the server, if the default server found in uf80c.ini is incorrect. The *server* value can be a hostname or IP address of the system running the UnForm server, and may optionally include :*port* suffix, such as ourserver:27280. The port can also be specified with the –port option.

-shift *n*	Causes all input text to shift *n* columns to the right, similar to the action of the shift command. This can be useful in conjunction with the –x crosshair option to force text to match the alignment it would have with a shift *n* command in a rule set.
-slon "*codes*" -sloff "*codes*"	Causes local (client side) output to be started with the slon *code* and ended with the sloff *code*. This option is only supported in the UNIX client. The *code* can contain text and special escaped characters: \e Escape \n Newline \r Carriage return \0*nn* Octal character *nn* (i.e. \033 is an escape) \x*hh* Hex character *hh* (i.e. \x1b is an escape) These values are typically set in conjunction with a **–o /dev/tty** option, in order to send a job back to the client-side terminal device for slave printing. Use of these options also causes the UNIX client to attempt to change the stty setting of the –o device to "raw" for the duration of the output. A typical slave print client command line might look like this: `cat sample1.txt | uf80c –f simple.rul –slon "\e[5i" –sloff "\e[4i" –o /dev/tty`
-status -nostatus	Overrides the default behavior of the status window when submitting jobs in the Windows client uf80c.exe. The default behavior is to show the window for jobs that will be returned to the client, and not show the window for jobs that will be printed by the server.
-sshost "*server*:*port*"	Sets the support server host and port, overriding the sshost= and ssport= lines in uf80d.ini for the job. See also the sshost() code block function.
-textjob	If this is specified, then the textjob$[all] array is built even if that is not the default setting, as defined in textjob=*x* in uf80d.ini.
-testpr *font symset*	Generates a test print showing nearly all characters (ASCII 1 to 254) in the *font* and *symset* codes identified. For a list of font codes and symbol sets, see the ufparam.txt file, sections [fonts] and [symsets], respectively. This option supports both laser and pdf drivers. To generate a PDF file, add "-p pdf" to the command line. Output can be sent to a file or device with the "-o" option, or on UNIX can be piped to standard output. Note that with the pdf driver, the only symbol set used is 9J.
-timeout *n*	Sets the socket timeout, for connecting to the server, to *n* seconds. If the server takes more than this amount of time to accept the connection, the client produces an error. The default value is 10 seconds.
-trans "*filename*"	Specifies a translation file, used to substitute text, barcode, and search values that are dynamically translated as the job runs. The active file can also be specified with the settrans("*filename*") code block function.

-usess	Forces the use of the Windows Support Server for image conversions and Ghostscript execution, even if server-based configuration is enabled.
-v	Causes UnForm to print version information and exit.
-vshift *n*	Causes all input text to shift *n* rows down, similar to the action of the vshift command. This can be useful in conjunction with the –x crosshair option to force text to match the alignment it would have with a vshift *n* command in a rule set.
-wait	Causes the client to wait for job completion, even if the server is printing the job. Normally, when the client submits a job to the server, it will exit as quickly as the server acknowledges the job has started (not, of course, if the output needs to come back to the client). By including the –wait option, the client will wait until the server job is complete, even if the output will be handled by the server. The purpose of this option is to allow client reporting of any errors the server might encounter once the job starts.
-x [*page*[,*page, …*]] -xl [*page*[,*page, …*]]	Causes the first page of input, or the pages specified, to be printed with a cross hair pattern. This is typically done once to assist in determining placement of text, and then removed. Sometimes, a special printer definition is set up within an application, using the -x option, so that any form can be printed to that printer for layout purposes. Note that setting the environment variable UFC to "y" will cause this option to be automatically implemented. Optionally, specify one or more (comma-delimited with no spaces, or hyphenated for ranges) page numbers to get UnForm to produce cross hair patterns on specific pages of the input stream. For example, '-x 1,3-5' would produce cross hair patterns on pages 1, 3, 4, and 5, suppressing all others. If the input doesn't contain form-feed page delimiters, be sure to use the –page option as well. When the –x option is used, no rule set is applied to the job. See the **crosshair** command if you want to apply a grid to enhanced output. The –xl option will produce a landscape version of the crosshair printing.
-z *filename* -zx *filename*	Adds command line options contained in the text file *filename* to the command line as if they were part of the command line itself. This option is helpful if a command line length exceeds the operating system limit. If the –zx option is used, then *filename* is erased once it has been read. The file is simply a text file with arguments separated by white space or new lines. Lines beginning with a # character are not included.
Job Status Viewing Options	

-jobs -myjobs	These options trigger the viewing of jobs submitted to the server. The –jobs option shows all jobs submitted to the server, while –myjobs shows just those jobs submitted by the current user. Job records are kept for a configurable amount of time, determined by the age= setting in the uf80d.ini file on the server. By default, the data displayed includes the job number, date/time, user, input size, pages complete, percentage complete, and status. The –detail option, below, adds the rule set, driver, and error message columns. The output can be sent to standard output on Unix, and filtered, like this: uf80c –jobs | grep 'Errored'| more On Windows, either a –o option or a –p winpvw option is required, and the output goes to the named file or to a temporary file and viewed with that file types native viewing application.
-active	This option will limit the job display to jobs that are currently processing on the server.
-detail	This option will cause the job listing to include additional data, including the ruleset, driver, and any ending error message.
-jobsfmt *format*	Controls the format of the jobs output. Two options for *format* are supported: txt and csv. The txt option produces a tab-delimited list, while the csv option produces a comma-separated-values list. The default format is txt. On Windows, when used in conjunction with –p winpvw, the native application for the format (or output file specified) is used. On systems where Microsoft Excel is the default application for .csv files, specifying –jobsfmt csv –p winpvw will result in a display of jobs in Excel.
Archive and Document Management Options	
-arcargs "*args*"	Sets the default archiving subjob options. When archives are written via UnForm jobs (as opposed to the –arcput option), a PDF file is generated as a subjob while the UnForm job is processed. The subjob may require UnForm command line options to run properly. This option sets those options. The **archive** command can also set options.
-arccats "*cats*" -arccategories "*cats*"	Sets the archive document category indexes for document writing. There can be any number of semi-colon delimited categories, with each category supporting up to three pipe-delimited levels. Each level can be up to 20 characters long. A different character than the pipe can be use by specifying a –arcsep option. Generally this option must be enclosed in quotes to protect semi-colons, pipes, and spaces in the command line. The format structure is this: "cat1.1|cat1.2|cat1.3;cat2;cat3.1|cat3.2;..."

-arcdel	Triggers removal of an archive image or document, honoring the options for -arclib, -arcdoctype, -arcdocid, -arcsubid. Delete access to the library is required for the –arclogin user. If a –arcsubid is supplied, then just that image is removed. If no –arcsubid is supplied, then the document record and all sub-images are removed.
-arcdocid "*docID*"	Sets the archive document ID for document writing or retrieval.
-arcdoctype "*doctype*"	Sets the archive document type for document writing or retrieval.
-arcdtm "*ymddate*"	Sets the archive document date for document writing to the date specified. Normally this value is calculated automatically. The date supplied must be in the format yyyymmddhhmmss, with the first 8 characters required.
-arcdtmupdated "*ymddate*"	Used with the –arcsearch command to filter the date/time updated. Setting this does not affect the date updated stamp when writing archive records.
-arcend "*end*"	Specifies an ending point for –arclist and –arclistdocs options, allowing for a range a range of documents to be returned. The ending point should logically be associated with the order specified in the –arcorder option. If the order is "category", then *end* can contain pipe separators to indicate category level breaks.
-arcentityid "*entityid*" -arcentid "*entityid*"	Sets the entity ID for document writing, either the default for UnForm jobs or the value for direct (-arcput) document writing.
-arcexists	Returns a 1 (found) or 0 (not found) plus a CRLF to the output device, typically –o client:*filename*, based on testing the existence of -arclib, -arcdoctype, -arcdocid, and –arcsubid options. This option does not require a login. The number of options supplied determines the test performed, from simply testing for library existence, to looking for any documents of a specified document type, to testing for a specific document ID, then finally image sub ID.
-arcfilter "*filter*"	Specifies a filter string to apply to the –arclist and –arclistdocs options. This filter applies to all elements of the document, including title, keywords, and categories. If the filter starts with a tilde (~), the remaining characters are interpreted as a regular expression. Otherwise, wildcard characters * and ? are supported for matching any characters or any single character.
-arcget	Triggers retrieval of an archive image, honoring the options for -arclib, -arcdoctype, -arcdocid, -arcsubid, and -o. Read access to the library is required for the –arclogin user.
-arckeywords "*kwds*"	Sets the archive document keywords for document writing. There can be any number of semi-colon delimited list of words or phrases. If none is set, then documents inserted by UnForm jobs (as opposed to those written via a –arcput option) will have a configurable number of unique keywords generated automatically from input file content. Generally this option must be enclosed in quotes to protect semi-colons and spaces in the command line. The format structure is this: "kw1;kw2;..."
-arclib "*libpath*"	Sets the archive library path for document writing or retrieval.

-arclink “*links*” -arclinks “*links*”	Sets the archive document links, which is a semi-colon delimited list of links to web documents, either external or other archive documents. Each link in the list can be in one of the following formats: • A full URL, optionally matching a URL used to load a document or image from a library, or a URL to an outside page or document. This structure, if it begins with http:// or ftp://, can be prefixed with a title in the format of *title=URL*. If the title is specified, that becomes the visible link in the browser. • A simplified pipe-delimited structure of *library*|*doctype*|*docid*[|*subid*], which is displayed in the browser interface as a URL link to the document or image named by library, document type, document ID, and optionally image sub ID. There can be any number of links in the list.
-arclist	Triggers a listing of archives, honoring the options for –arclib, -arclistfmt, -arcorder, -arcstart, -arcend, and -arcfilter. This listing displays the document archives, but not the image sub ID’s. Use the -arclistdocs for this additional information. The -o option can be used to send the results to a file.
-arclistdocs	Triggers a listing of archives and image sub ID’s, honoring the options for -arclib, -arclistfmt, -arcorder, -arcstart, -arcend, and -arcfilter. This listing displays the document archives plus their associated images. The -o option can be used to send the results to a file.
-arclisttypes	Triggers a listing of document types in the library specified with an associated –arclib option. The –o option can be used to send the results to a file. A -arclogin is typically required. The listing is a single text column of values, one document type per line.
-arclistfmt *fmt*	Sets the format for -arclist and -arclistdocs. It must be one of the following: tab, csv, pipe, html[:*stylesheet*], or xml|xmlf[:*stylesheet*]. If the format is xml, a XSLT stylesheet can be referenced by URL with a colon suffix: “xml:http://somewhere.com/stylesheets/mystyle.xsl”. Likewise, if the format is html, a CSS stylesheet URL can be referenced with a colon suffix: “html:http://somewhere.com/default.css”. An alternate format, “xmlw3c:*stylesheet*” can be used to generate the header type= tag as “application/xsl” rather than “text/xsl”. If stylesheets are specified, the application that views are parses the document must have access to the specified stylesheet. If the format is xmlf, then a base64-encoded version of the image file is included with the xml output.
-arclistlibs	Triggers a listing of libraries, including path, description, creation date, and default permissions.

-arclogin “*userid*/*pswd*” | ask	Sets the login user ID and password (they must be separated by a slash - /) for the command. Logins are required for many archive command line operations. Under some circumstances, login information can be read from files. See the Document Archiving and Management chapter for more details. Optionally, enter the word “ask”, or provide no argument at all, and the client will prompt for the login information.
-arcnotes "*notes*"	Sets the archive document notes for document writing. To force line breaks, insert \n mnemonic character sequences.
-arcorder *order*	Sets the order of the lists retrieved by the –arclist and –arclistdocs options. The order must be one of the following: id, date, title, or category. The default order is id.
-arcput	Triggers writing of a file directly to a library, bypassing UnForm processing of the input file. The option requires values for -arclib, -arcdoctype, -arcdocid, -arcsubid, and -i, and supports the archive property setting options such as -arctitle, -arckeyword, etc. Review the archiving chapter for more details.
-arcsep *char*	Sets the separator character for category levels to *char*. The default is |.
-arcsearch	Triggers a search to be performed, in conjunction with the -arclib, -arcdoctype, -arcdocid, -arctitle, -arcentityid, -arcdate, -arckeywords, -arcnotes, -arccats, -arclinks, -arctext, and -arcsubid option. The search results are sent to the –o output file. The format of the results is controlled by the -arclistfmt option.
-arcsearchdocs	Identical in function to –arcsearch, except that image sub ID information is additionally returned.
-arcstart “*start*”	Specifies a starting point for –arclist and –arclistdocs options, allowing for a range a range of documents to be returned. The starting point should logically be associated with the order specified in the –arcorder option. If the order is “category”, then *start* can contain pipe separators to indicate category level breaks.
-arcsubdtm “*ymddate*”	Sets the archive document image sub ID date for document writing to the date specified. Normally this value is calculated automatically. The date supplied must be in the format yyyymmddhhmmss, with the first 8 characters required.
-arcsubid "*sub ID*"	Sets the archive document image sub ID for document writing or retrieval. When used in conjunction with –arcget, you can use two special suffixes, “-<” and “->”, to return the first or last sub ID that matches the sub ID string. For example, a value of “@UnForm->” would return the last sequential @UnForm sub ID in an auto-sequenced library. Escape the suffix as “\->” to look for the literal value.
-arcsubtitle "*title*"	Sets the archive document image sub ID title for document writing.
-arctextdata “*value*”	Used in conjunction with the –arcsearch option to search text image data, archived with @text sub IDs, for the value supplied.
-arctitle "*title*"	Sets the archive document title for document writing.

Two additional options, -about and –configure display client version information and a configuration dialog, respectively.

UNFORM AFO – APPLICATION FORMATTED OUTPUT

UnForm recognizes PostScript and PDF input streams, and will attempt to process them as pre-formatted print jobs. To do this, both GhostScript and a Windows Support Server must be configured and available, as to parse out the text from these jobs, UnForm converts the job to PDF pages, and then uses a feature in the Windows Support Server to locate all the text elements found in the job.

> Caveat: Note that not all such print jobs contain text. Some contain images of text, and some contain a mixture of text and images of text. Only true text data can be used by UnForm as rule set data. In addition, sometimes text elements contain large regions of clear space around the text itself, posing some challenges for parsing text by location. The availability and usefulness of text is determined by the printing application and GhostScript, not UnForm.

The Design Tool can submit PostScript or PDF sample data to the UnForm server, and highlight each text element found on each page (by checking the Add Text Base option on the Preview menu).

The technique used by UnForm when it receives a PostScript or PDF print stream is to generate an overlay of each page in the output format of the job. UnForm graphical commands can then be used to add elements to this overlay, to scale it, and to erase regions from it. Other than selective erasure, it is not possible to modify the overlay. In many cases, there will be very limited, if any, cosmetic enhancements needed, allowing the implementor to focus exclusively on document management features such as electronic delivery and archiving.

The most common method of integrating PostScript input with UnForm is to use a Windows printer configured with a PostScript print driver and a TCP/IP port directed an UnForm TCP/IP monitor.

The –noafo command line option can be used to suppress AFO processing for Postscript jobs, and may also be useful as an argument passed to a subjob in a jobexec() function, as subjobs of AFO jobs are by default treated as AFO jobs themselves.

As the PDF pages are treated similar to overlays, the orientation of the UnForm job must match that of the PostScript or PDF input. For example, if the input uses landscape orientation, the UnForm rule set should include a landscape command.

Text vs. PostScript/PDF Print Stream Management
When working with plain text input, UnForm has commands that manipulate or apply enhancements to a text print stream, such as font, bold, and erase. Also, code blocks can manipulate the text$[] array, resulting in modified print stream text. However, when working with PostScript print streams, there is no text array, and commands that depend on it are not available. One exception is **erase**, which is translated to be a shade command with a shade value of 0, resulting in erasure of the specified region of the overlay. Also, the **notext** command and its new synonym, **nooverlay**, may be used to suppress printing of the overlay on any copy or all pages.

The following commands are not compatible with PostScript input:

- across
- bold
- down
- font
- hline
- hshift
- italic
- light
- move
- page
- shift
- underline
- vline
- vshift
- any Zebra- or label-only commands

In addition, many commands support anchor text or patterns, which cause a search of the text content of the page to locate positions to apply enhancements. Supported commands that offer this feature, such as **barcode**, **box**, and **text**, continue to support the anchor search technique. However, since the location of PostScript text regions do not always correspond to the visual location or size of the text, accuracy can vary.

If PostScript text regions vary from visual location or size, then detection logic may require greater flexibility than with simple text input streams. The **detect** command has been enhanced to support partial columns and rows, but it may be necessary in some cases to detect elements from the whole page rather than regions.

Text Array Limitations in Code Blocks and Expressions
Many code block functions that work with a text print stream are also not available. However, the get() and mget() functions have been enhanced to return text data from the PostScript print stream, plus three new functions have been added, gtextcount(), gtextitem(), and gtextfind(), which provide access to the text elements parsed from the PostScript print data. A new variable, nooverlay, can be set to 1 in prepage or precopy code blocks to suppress the printing of the overlay. This can be used to manage multi-format jobs, such as those with terms and condition attachments.

The following code block functions are not compatible with PostScript input:

- cut()
- delpage()
- getpage()
- inspage()
- mcut()
- mset()
- putpage()

- set()

The arrays text$[], textjob$[], and textpage$[] are not available.

New Functions For Accessing Text

- gproperty() returns values from PostScript DSC comments in the print stream.
- gtextcount() returns the number of text elements in a page.
- gtextitem() returns text and optional region information for a given text element on a page.
- gtextfind() searches for patterns in text and returns arrays of text and region information found.

FLOW OF PROCESSING

UnForm processes jobs in a complex but defined manner. The following list describes in general what occurs when a job is submitted:

The client program is executed with options, generally including input and output specifications, a rule file, and any other command line arguments. On UNIX, it is possible for the input and/or the output to be "standard input" and "standard output", so that the client can process jobs in a pipe. Here are a few examples:

```
uf80c –i sample1.txt –o ">lp –dlaser –oraw" –f acme.rul

cat sample1.txt | uf80c | lp –dmylaser –T pcl

cat sample1.txt | uf80c –p pdf >/home/mypdfs/xyz.pdf

uf80c –i sample1.txt –o client:myfile.pdf –p pdf
```

In all cases, the input file comes from the client and is sent to the server. With a –o option, the output normally stays on the server, though if the output designation is prefixed with "client:", then it is returned to the client. On UNIX, if "standard output" is designated, the output is also returned to the client. The rule file specified with the –f option resides on the server.

For performance reasons, it is normally desirable to specify a server-based output designation with the -o option. In that circumstance, the client only runs long enough to submit the job and ensure the command line arguments are acceptable to the server, then returns to the application. If the client will be receiving the output, it must wait for the job to finish and retrieve it, which can be time consuming (though certainly less so if the client and server are on the same machine).

When the server receives the job, it stores the input in a temporary file, and calls the UnForm processor to handle the job.

UnForm reads the input file to obtain the first page. It looks for a form-feed, or if no form-feed is found, it reads the first 255 lines. It then strips the data of any PCL escape sequences in order to get a plain text array of lines. Lines must be terminated with line-feed characters (ASCII 10), carriage-returns characters (ASCII 13), or carriage-return, line-feed sequences (ASCII 13, 10).

Note that if the input is found to be PostScript, then UnForm processes the job using UnForm AFO.

This first page is processed against the rule file. If a –r *ruleset* command line argument was used, then the rule file is scanned for the specified rule set. Otherwise, each rule set's detect statements are tested using the first page of text. When the rule set is found, it is parsed into commands and code blocks. If no rule set is found, then the job is handled by pass-through logic, or if a rule set was specified with –r and not found, an error occurs and the job exits.

If the parsed rule set indicates a page size with the page *n* command, any excess lines read from the first page are returned to the input buffer. As the input stream is read for additional pages, UnForm will read only *n* lines per page. Note that if a form-feed character is encountered before *n* lines have been read, then the page is also considered complete.

If a prejob code block is present, it is executed.

Now processing of the job begins. Each page is processed in the following order:

- The prepage code block is executed.
- Any command expression values are resolved.
- For each copy:

    - The precopy code block is executed.
    - Command expressions are resolved.
    - Any hshift or vshift commands are executed (if shiftfirst=1 in ufparam.txt [defaults]).
    - Move commands are executed.
    - Font, bold, italic, underline, and light commands are executed.
    - Shade commands are executed.
    - Box commands are executed.
    - Text commands are executed.
    - Hline and vline commands are executed.
    - Erase commands are executed.
    - Any hshift or vshift commands are executed (if shiftfirst=0 in ufparam.txt [defaults])
    - Attach commands are executed.
    - Image commands are executed.
    - Barcode commands are executed.
    - The application text, with any font attributes applied, is added.
    - Micr commands are executed.
    - The postcopy code block is executed.

- The postpage code block is executed.
- When all pages have been processed, the postjob code block is executed.
- As the job is processed, the output designation for each copy is checked, and if the output is changed, predevice and postdevice code blocks are executed. When running a PDF job, the only time the output can be changed is in the prejob code block, or with an output command that is non-copy specific. The postdevice code block is executed after the output is complete and closed, making it suitable for handling the output file itself (for emailing, faxing, etc.).

Once the job is complete, it is available to return to the client, if the client's command line requires it. The client has monitored the job for completion in that case, and it then retrieves the job output. Note that if the rule set has overridden the output designation for the job, or part of the job, then the client will only be able to retrieve what was sent to the original output designation.

So the following scenario will conflict:

- **uf80c –i sample1.txt –o client:/tmp/invoice.pdf –f advanced.rul –r invoice**
- In the invoice ruleset is this: **output "/home/pdfs/invoice.pdf"**
- The server will send output to its /home/pdfs/invoice.pdf file, leaving the temporary output for the client empty. The client /tmp/invoice.pdf file will be an invalid, empty file.

CONCEPTS, PRIMER, AND TIPS

UnForm is a very powerful tool, with dozens of commands and features. It can be difficult to grasp the basics from such a large toolset, but the basics are really very simple. Once UnForm is installed by an administrator, the only skills required to develop typical business forms are an ability to edit text files on your system, and an ability to execute UnForm as needed to test your changes. The Windows-based UnForm Design Tool can provide an efficient environment for rule file development, if desired.

Here are some basic concepts that you should understand before proceeding:

- UnForm processes text input and produces formatted output. The input can come from a file or, on UNIX, can come from UnForm's standard input. The output can go to a file or a device on either the server or the client, or on UNIX can go to the client's standard output.

- UnForm can also process pre-formatted application output, if that output comes to UnForm as PostScript. See the UnForm AFO chapter for more information.

- UnForm uses a *rule file* to define all the form and print jobs it might process. In that rule file are one or more *rule sets*, each of which represents one form or print job. Rule files and the rule sets they contain are simply text files with command lines, which you can edit with any text editor. The rule file should be stored in the UnForm directory, and specified with the "-f *rulefile*" command line argument. If you don't specify the rule file on the command line, then the default rule file named at installation is used.

- Unless the "-r *ruleset*" command line option is used, UnForm reads the first page of input and compares that first page with all the **detect** statements found in each rule set. These statements instruct UnForm to look for text or patterns at specified locations or lines (or anywhere on the page). If all the detect statements for a given rule set match the contents of the first page, then UnForm selects that rule set and begins to produce output. If a match is not found, then the next rule set is tested, and so on until all the rule sets have been tested. If no match is found, then UnForm will pass the job through without any changes or enhancements, or in the case when a pdf or pcl driver is specified with a **–p** *driver* command line option, will produce a text job scaled to fit each page.

- Each job has its own *geometry*, that is, the basic columns and rows to which UnForm scales everything. If you specify **cols 85**, then UnForm will scale each character and all the enhancement positions and sizes to 1/85th of the printed space between the margins. In a sense, the job wraps enhancements around the text input as it is sent to the output.

- The commands in the rule set determine what enhancements are applied. These can be text additions, font changes, boxes, shade regions, barcodes, images, and more. Each change is controlled by a command line in the rule set, such as **box 5.5,2,20,4**.

  Some commands don't add output, but instead modify the text input to UnForm. The text will normally print in the Courier font, scaled to the number of columns you specify. You can change the attributes of that text in any rectangular region with **font** command, or manipulate it with the **move** and **erase** commands.

- Some commands control the printer. For example, the **tray** command can select the input tray on a laser printer, and the **bin** command can select an output bin.

- You can have UnForm generate multiple copies of each page of input. Each copy can have unique characteristics by using **if copy *n*** blocks. This is a simple structure that starts with a line "**if copy *n***", where *n* is the copy number, followed by any number of lines of enhancement commands, followed by a line "**end if**".

Creating Rule Files with the UnForm Graphical Designer

- Obtain sample output from your application for the form you want to design. Most applications provide the means to print to a text file. If no other means exists, you can define a printer that prints to UnForm with a **–debug** command line option, in which case UnForm will leave a copy of the input stream on the server, under the UnForm directory, in temp/*jobno*.in. You can find job numbers and their print times and size with the **uf80c –myjobs** command.

  Store this text file in the UnForm directory on the server.

- Start the UnForm Designer on a Windows system, and connect to the UnForm server when prompted. Create a new rule file, then a new rule set, then set the sample to the file created above. The UnForm Designer is a rule file editor with on line help, command editors, and drawing and preview capabilities. More information about using it can be found in the on line help that comes with the product.

Manual Rule Set Creation Steps

- Obtain sample output from your application for the form you want to design. This output can be printed to a text file, or you can simply use two printers defined with UnForm, one with the crosshair option (-x), the other with normal output. If you are working on a Windows system or have network access from a Windows system to the server where UnForm operates, you can use the pdf driver and an Acrobat Reader to save paper while developing the design.

- Print your sample through UnForm with the crosshair option turned on. This will provide you with a grid of text positions printed by your application. If you have a file printed by your application, the command line for a grid would look like this: uf80c –x 1-99 –i *input-file* –o *output-device* or uf80c -x 1-99 –i *input-file* | lp -d*xxx* . If your sample does not contain form-feeds, you can add a **–page *n*** option to tell UnForm how many lines are to be read per page.

- Since you will be printing this sample many times, you may wish to create a script or batch file to automate the command line, which will be something like: uf80c –i *input-file* –f *rule-file* –o *output-device* or uf80c –i *input-file* –f *rule-file* | lp -d*xxx*.

- Looking at the text of the input file, determine what makes this job unique. Sometimes there is a title, such as "PURCHASE ORDER", printed at a specific position. That may be enough to determine the uniqueness of the document so just add detect *column, row*, "PURCHASE ORDER". You might need to find multiple patterns by using more than one detect statement. Patterns are specified by starting the detect string argument with a ~ character. The balance of the string is a regular expression. Common syntax elements for regular expressions include "." to match any character, [0-9] to match any digit, [A-Z] to match any capital letter, and * to match any number of repetitions of the prior match character. A more complete description of regular expressions is in the Regular Expressions chapter.

  To try out your detect statement(s), try adding just those statements plus a single text command, then print the job. If your job prints with that text in addition to the text from your application, then your detect statements are working. This is what the rule set will start to look like:

  ```
  [purchase_order]
  detect 40,2,"PURCHASE ORDER"
  text 1,1,"Test Text"
  ```

  Note that it is possible to execute a rule set without detect statements, by adding "-r *ruleset*" to the command line.

- The rest of the form design is simply a matter of adding commands for text, boxes, and shade regions. It is usually best to work consistently from top to bottom, left to right in the different sections of the form. Use comments (lines starting with #) liberally; they make the rule set easier to follow when you come back later to make a change.

A good place to see complete rule sets are the sample rule files provided with UnForm, simple.rul and advanced.rul. These two files are thoroughly documented in Sample Rule Sets chapter. In addition to simple form designs, the samples show techniques with complex designs, such as jobs with multiple formats of input, and jobs that have embedded programming capabilities.

Tips and Techniques

- Always start with a crosshair pattern, so the basic text provided by the application, and its exact placement, can be seen. As the crosshair mode prints just the first page, use short versions of the reports or forms. There are several ways to create a crosshair version of a report:

    - Print the report to a file, then process that file with UnForm's command line, such as **uf80c -i *filename* -o *output-device* –x**

    - Add a printer configured with the "-x" option, and print to that printer.

  If your report doesn't contain form-feed characters at the end of the page, then you should print just one page worth of data, or add a **–page *n*** option to the command line. Otherwise, UnForm will assume the page is made up of as many lines as are printed, up to 255 lines.

- Use **detect** statements to identify each form. UnForm is designed to process all your reports and just enhance those it can identify; all others are passed through unchanged. This is easier to set up than forcing a given printer device to be named for every form or report, as is required of most form packages.

- Specify the columns and rows for the form or report using the **cols** and **rows** commands. If this isn't done, then UnForm will assume 80 columns by 66 rows. An exception to this assumption is that if a **page** keyword is used, then the rows will be taken to be that value unless a rows command is also present.

- Remove unwanted text with the **erase** command, or move it with the **move** command. In programming code, such as in the prepage or precopy routines, you can modify the text$[] array directly or via the set() function.

- Apply attributes to the text with the **bold**, **italic**, **light**, or **underline** commands. These apply to the text generated by the application (not to text you add with the **text** keyword). Or use the **font** command, which can apply any of these attributes as well as apply other characteristics to the application text data.

- Use the **font** command to modify the font of text from the application, All text printed by the application will print in Courier unless changed with the font command. When changing to a proportional font, be sure to make the changes to specific logical regions, such as a column of prices. If you change the font for the entire page, then columns will not align properly.

- Add text, such as headings or messages, with the **text** command. Text can be literals enclosed in quotes, named values from a substitution file if prefixed with "@", environment variables prefixed by $, or an expression enclosed in { } characters. Text can be rendered at any size and in any font supported by the printer or device. Remember that fixed pitch fonts, such as Courier, are sized in characters per inch, while proportional fonts are sized in points. The larger the cpi, the smaller the font. The larger the point size, the larger the font.

- Add shading and box drawing with the **shade** and **box** commands. Reverse shading is accomplished by shading a region with 100% (black) shading, and using a **font** or **text** command to modify the text to shading of 0% gray (white). Simply using a row or column value of 1 will draw lines. To draw a box and shade the interior, use the shade option of the **box** keyword.

- Add logos and other images with the **image** command. With this command, UnForm normally looks specifically for PCL raster images (or PDF images if the pdf driver is used) in the file. UnForm can also be configured to use Image Magick or Image Alchemy for on-the-fly conversion of traditional image formats to native PCL or PDF.

- Use the **attach** command to add overlays or attachments. This command does not search only for image data. It does, however, search for and remove initialization and form-feed codes.

  Attachments should be treated as separate copies: use the **pcopies** command to allocate enough copies, then use **if copy *n*** to add the attachment, **notext** to suppress the application text output, and make sure other enhancements don't apply to the attachment copy.

  To create an overlay, use the **attach** command, but allow the text and enhancements to also be applied on the same copy. Attachment documents for PCL output can be created using a PCL5 printer on Windows, selecting the Print to File option or setting it up to use a FILE: port. For PDF attachments, use Adobe Distiller, choosing non-optimized, ASCII output options.

- If the application doesn't use form-feeds at the end of each page, then use the **page** keyword to tell UnForm how many lines are used for each page. Many applications, especially with forms, will use just line-feeds when scrolling to the top of each form. UnForm will need to be told where the end of a page is, in this case.

Use Business Basic programming as a powerful macro language. All the data that is sent by the application to each page is available for your use. Use this data to get fax numbers and generate faxed copies, or to print shipping labels derived from the invoice ship-to addresses while packing lists are printed, or to add additional information such as costs or comments to forms, or to print logs or send email. See the precopy{} command reference, and the Programming Code Blocks chapter for more information.

DOCUMENT ARCHIVING AND MANAGEMENT

Overview

The UnForm document archiving and management component provides a suite of archiving functions which are seamlessly added to UnForm's library of commands and tools for document enhancement and delivery.

Existing UnForm integrators and designers familiar with UnForm's unique text filter technology will find it simple, intuitive and hassle-free to add archiving commands and arguments into UnForm's rule-file oriented flow of processing. Or rule-files can be bypassed altogether to archive non-UnForm-generated documents using the familiar command-argument interface to the UnForm client software. And UnForm's separate Scanning workstation component can add scanned image files to the archive and match them with existing documents previously stored using manual ID matching or barcode/OCR-based image ID capture.

With a universal, browser-based document retrieval interface, UnForm makes it easy to browse, search, list, view, administer, and secure archive libraries. Libraries can scale up to a theoretical capacity of 4-billion documents. Context-sensitive help links include sample page images, and help guide the user or administrator through the browse, search and administration functions. Sample archives are included and are referenced in the help pages. They can aid in the design of a logical custom archiving library and identification structure suited to the needs of sophisticated end-users.

Flexible pre-defined and user-defined document index structures are designed to make document identification and retrieval practical, fast and easy. Pre-defined index structures exist for a two-segment type-ID index, and a date-time index. A user-defined up to ten-segment pipe-delimited category key structure is also provided for indexing. The browser-based document retrieval interface provides an intuitively sensible drill-down browse function through the levels of the multi-segmented indexes. Libraries are file-system-based locations. A three-tiered library-document-image hierarchy is employed which allows multiple versions of a document, e.g. text and pdf, to be stored together, uniquely identified by a Sub-ID index, and further allows multiple text or non-text image or data files to be attached as sub-documents to a parent. When archiving from an UnForm job, both text and pdf versions are stored automatically.

Subject to access-rights, document and images being listed and/or viewed in the browser interface can have properties modified by users to update document status, correct indexes, and maintain associated notes and keywords at a document level. Files on the network can be browsed and added as sub-documents from within the browser.

Security is managed by library and by user and/or group. All documents are encrypted and compressed when stored in the library. To access documents, a user login is required, and each login can be granted read, write, or delete access to a given library, or can be allowed to access the library based on the library's default access profile.

Groups may be defined by the administrator. Users can be assigned to one or more groups, and library access can be granted to a group rather than individual users. If a user is granted specific rights to a library, those rights are used. Otherwise, the list of groups that the user is a member of is scanned and all rights offered to each of those groups are granted to the user.

Note the following change from UnForm 7.1: if a user is assigned to one or more groups, default library permissions are not applied. In Version 7.1, default library permissions were applied in addition to group membership permissions. This meant that if the default library access was Read, a user would have access to a library even if all assigned groups were denied access. This change gives group membership precedence over default access, and provides the administrator more explicit control over library access.

A user can also be assigned an Entity ID. This classifies the user as an *external user*. External users are given very limited access in the web browser. They can only browse records by document type and ID, or by date. Access is limited to documents that have a matching Entity ID value.

The browser-based, multi-library search function creates disk-based query-lists of documents which can be further manipulated independent of other documents in the library. The query lists can be the basis for what are known as bulk actions, which include copying to new or existing libraries, transferring to new or existing libraries, and exporting to HTML. The HTML export produces a completely self-contained, pure-HTML directory structure suitable for loading on other storage media, such as a CD/DVD, a zip file, a web site directory, etc.

Imagine, for example, exporting all of a customer's invoices from a date range to a zip archive and emailing it to them. Another example would be to off-load old documents to external storage, then purging them to free up disk space.

The separate Scanning workstation client provides image management and uploading into a library. Images can be scanned or imported from the PC's file system. Both barcode recognition and OCR recognition assist in automating document identification. Using VBScript, a developer can automate the interpretation of such data and use database access or other coding logic to generate document property and indexing information.

Structure Details

The structure of UnForm archiving is a hierarchical one, where an image of a document is at the bottom of a nested structure. In fact, what you may consider a “document”, such as an invoice or a purchase order, is at the middle of this structure, as there can be many versions of a given document. Here is a description of this structure:

Library	A *library* is a folder or directory path to the location where archived documents are stored.

	There can be one or many libraries to store documents. Access and security is controlled by library, so the process of designing a single or multiple library structure, and determining which documents will be stored in which library, needs to take into account the access rights of groups and users. Beneath a library path UnForm uses a file storage algorithm with a theoretical capacity for over 4 billion distinct documents, each of which can contain multiple images. All data about documents and images, as well as the images themselves, is encrypted and compressed.
Document	A *document* is one or more related files which share a unique *Document Type* and *Document ID* combination, different from any other document in the archive. Because the document's "source" data itself is stored in what UnForm recognizes as an "image" file, the document unit in UnForm can be seen as an "envelope" or "wrapper" which surrounds one or more associated files.
Image	An *image* is a single document file with a distinct ID (called a sub ID) which distinguishes it from other files associated with the same document. An image can be any type of file, not limited to image files. For example, when UnForm adds an archive from an UnForm print job, it adds both text and PDF images, with ID values of @text and @UnForm. Think of a "text image" of an invoice, versus a "PDF image" of the same invoice, versus a signed delivery slip "image file" pertaining to the invoice, versus a Word document "image" of the order quotation preceding the invoice, all associated with the same document ID. Each image within a document is identified with a unique *sub ID.*

Document-level Identification

The table below briefly describes the EIGHT main data elements which UnForm uses to identify documents at the document level. With the exception of the date/time stamp, and some character-separator rules enforced by UnForm on some of the fields, the data format for each of these text elements is user-configurable. A significant part of the administrator's implementation process is to design a document identification structure for the archive which will meet the enterprise's needs over a meaningful period of time.

Document Type	• First segment of the primary document identifier-key. • Maximum 20 characters • Null value allowed

Example document types from our sample libraries:

demo_sales	demo_accounting	demo_purchasing
"ArStatement" "OpInvoice" "OrderPickQuote"	"ApAging" "ApCheck" "ArAging" "GLDailyDetail" "OpSalesRegister"	"PurchaseOrder"

Document ID

- Second segment of the primary document identifier-key.
- Maximum 20 characters
- Null value NOT allowed. However, the null is trapped by UnForm and replaced with a 10-digit unique serial sequence number. Note that the number sequence is global within a library, but not within a document type.

The combination of Document Type and Document ID form a unique identifier-key to a document within a library.

Note that libraries are distinct units to each other so document identifier-keys are only unique to a library, in other words, identical document identifier-keys can exist in two distinct libraries without over-writing.

Categories

Secondary sorting values known in some software applications as sort-keys.
Example Categories from the demo_sales library include:

OpInvoice and OrderPickQuote Doc Types	ArStatement Doc Type

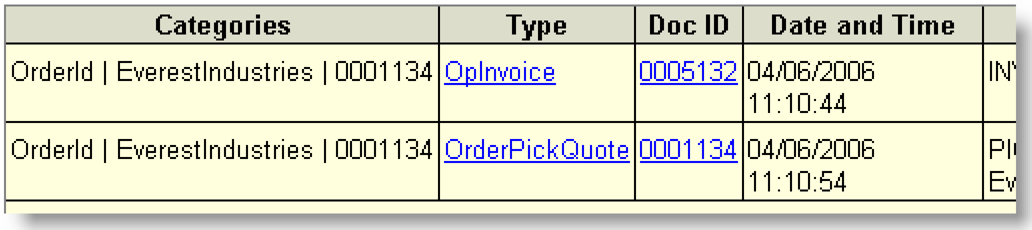
"Customer" | {CustName}
"Salesperson" | {SlspName}
"CustPO" | {CustName} | {POnumber}
"OrderId" | {CustName} | {OrderId}
	"Customer" | {CustName}

In the above samples, the text between quotes is literal, and the text between curly braces { } indicates a variable data field where the values for the document being archived are supplied by the application.

The pipe symbol | is used to delineate segments in a category, which allows UnForm to structure a drill-down presentation when browsing for documents. A category can contain up to ten pipe-delimited segments. The first segment should normally be a literal text category name, as shown in the example, to facilitate the category-type browse method when looking up documents.

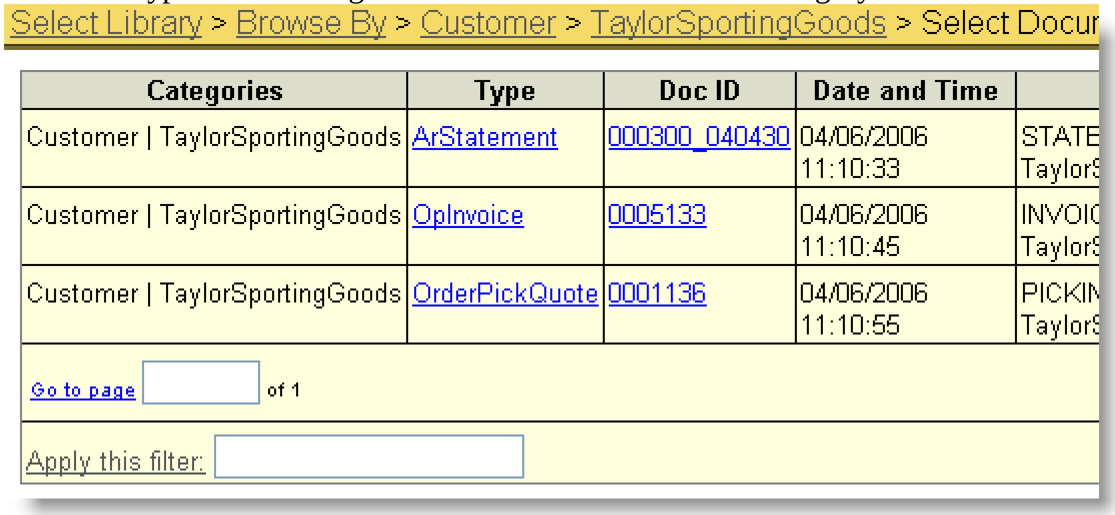
Note from the first column header above that the "OpInvoice" and "OrderPickQuote" document types are configured with the same list of categories. This is because in our sample database they are related documents in a relationship where an "invoice" is always preceded by an "order".

As an example from our samples of some of the considerations in choosing document identification strategies, by structuring a category on the invoice to reference the source order ID number, the following sample document browse list was enabled where an invoice and its related order document appear together in a list.



Categories	Type	Doc ID	Date and Time	
OrderId | EverestIndustries | 0001134	OpInvoice	0005132	04/06/2006 11:10:44	IN
OrderId | EverestIndustries | 0001134	OrderPickQuote	0001134	04/06/2006 11:10:54	PI Ev

The next example shows a browse list which has been started using a category browse method and drilling down from customer to customer name "Taylor". Notice the three different types of documents which were located for Taylor, because the different document types were configured with at least one identical category.



Select Library > Browse By > Customer > TaylorSportingGoods > Select Docur

Categories	Type	Doc ID	Date and Time	
Customer | TaylorSportingGoods	ArStatement	000300_040430	04/06/2006 11:10:33	STATE Taylor
Customer | TaylorSportingGoods	OpInvoice	0005133	04/06/2006 11:10:45	INVOI Taylor
Customer | TaylorSportingGoods	OrderPickQuote	0001136	04/06/2006 11:10:55	PICKIN Taylor

If a given category segment will potentially contain many thousands of items, it may be

	desirable to divide the segment into two tiers. For example, if a customer name is used as a segment, and there are thousands of customers, a two-tiered customer name could be designed, such as left(custname$,2) + "\|" + custname$. During browsing, the user would first locate the customer alphabetical group based on the first two characters of the name, then access just that sub-group of customers.
Document Title	A broad general description of the document, sometimes composed of several major data values that help distinguish the document from other similar documents.
Keywords	Additional document identifiers that can help narrow and limit searches to locate documents and groups of documents, improving search efficiency. Keywords are semi-colon delimited words or phrases. Often they are auto-generated from the content of the job submitted for UnForm processing. When keywords are auto-generated, the generation is subject to configuration rules found in the [archive] section if the uf80d.ini file.
Links	A list of links to other documents, either in the archive system or external to it. This list is displayed in the web browser interface when viewing the document. The list is semi-colon delimited, with each link being one of these formats: • A URL, such as http://acme.com, or a complete link to an UnForm browser page. If it starts with http or ftp, it can be prefixed with *title*= to specify a title for the browser to display. • A pipe-delimited structure that identifies the library, document type, document ID, and optionally image sub ID. The structure is *library*|*type*|*docid*|*subid*, with the |*subid* portion being optional.
Entity ID	A security data element which can be included with a document and/or user account to filter access to specific documents or groups of documents to login user accounts which carry access authorization referencing the same entity ID. The concept of the Entity ID is one of ownership, designed for situations where external web access to documents in a shared library needs to be restricted to the entity specified. For example, where customer XYZ can login and browse, list and view invoices for customer XYZ without ever seeing documents for other entities listed. If documents are written to the archive with an empty entity ID field, then any user account with an empty entity ID will pass the entity test for access to a record. In an environment with empty entity ID field on documents in the archive, simply assigning a user any non-blank entity ID value can be used to restrict access to all documents in the archive. At the current release level the entity ID field is treated, both for document properties

	and user properties, as a simple string of text. There are NOT provisions in the software for assigning multiple entity ID's to either a user or a document, nor any provision for entity ID sub-string referencing.
Date / Time	The date/time stamp of a document is used as a secondary sort-key in the library to allow a by-date browse-method drill-down to locate documents. The date/time value can be maintained by the UnForm rule set used to archive the document, or via a command line option. It defaults to the initial date and time the document was added. Additional **dateupdated** and **timeupdated** fields are maintained each time the document is updated.
Notes	Free-form text notes can be stored with a document and can be edited in the document properties box of a located document.

Image-level Identification

The elements discussed above apply to a document at a document "envelope" level, and serve to identify, group, and sort documents, and store additional useful information about a document, including free-form notes.

Because there can be multiple "images" associated with a document, each separate image file is assigned a unique identifier-key known as a Sub-ID. Note that the term “image” is used loosely here, and simply refers to a different version of the document. There can be a text image, a PDF image, and/or a tiff image of the same document.

When text-based documents are stored in the archive by an UnForm ruleset command, UnForm will default the Sub-ID value to @text for the text version of a document, and @UnForm for the pdf version of a document.

Documents stored in the archive via UnForm's command-line syntax will NOT have a default sub-ID assigned, so the user or the application must create a Sub-ID using the –arcsubid command line option.

In addition to the document file and the sub ID, UnForm also stores the date and time a particular image was last updated, plus a description field called a sub-title.

Adding UnForm-Generated Documents

UnForm document archiving supports several methods for adding documents to libraries. One of the most useful methods is via UnForm jobs themselves, through the use of command line arguments or an

archive command in a rule set. The benefit of this is that as jobs print and are formatted by UnForm, they can be automatically archived, eliminating the need to scan and archive reports using an external system.

Note: in order to be archived properly, jobs must be designed to successfully produce PDF output. In particular, jobs that use a pcl attachment or pcl images, but do not provide for PDF versions of these, will not be formatted properly in the archive.

If any –arc*xxx* command line arguments pertaining to the archive command are used, such as –arclib or -arcdoctype, the options establish defaults for any archive command found, or initiate job archiving as if an archive command were included in the rule set.

For example, assume the uf80c command line includes these options:

-arclib "/archives/reports" –arcdoctype "Reports"

If a rule set contains an archive command, the above defaults will be overridden by the command's options. However, if a rule set does not contain an archive command, or if no rule set is selected, the job will be archived regardless, using the above library and document type (in this case using an auto-generated document ID). This capability makes it easy to set up default archiving, with the ability to control archives on selected jobs with the addition of an archive command.

When UnForm archives a job, it evaluates all the elements of the archive command (or the values from the command line) page by page. Whenever an element changes, a new document is generated. In some cases, such as with hard coded command line options, these elements don't change during the job, and the whole job is archived as a single document. In other cases, an element such as a document ID might change as pages are processed, and a job can result in several documents being added to the archive.

UnForm documents can contain multiple versions or images, each identified with a sub ID. When UnForm archives one of its jobs, it archives two versions of the document. The first format is a PDF version of the document, which by default is given a sub ID of "@UnForm". The second format is a text version, derived from the incoming text stream. This is given a sub ID of "@text".

Archives generated from UnForm jobs receive automatic title and keywords if these values are not otherwise specified. If no title is specified and no **title** command is used, then the default title is derived from the content of the incoming text. Keyword generation is controlled with several parameters in the uf80d.ini file, including a maximum keyword count (keywords=*n*), a list of patterns to not archive (nonwords=*file*), and a list of characters to eliminate (nonchars=*list*).

If no document type is provided, then if a rule set is used for a job, its name is used as the document type.

The following command line arguments enable archiving and provide defaults for **archive** commands.

-arclib "libpath"

```
-arcdoctype "doctype"
-arcdocid "docid"
-arcsubid "subdoc ID"
-arcsubtitle "subdoc title"
-arctitle "title"
-arccats "cat1.1|cat1.2|cat1.3;cat2;cat3.1|cat3.2;..."
-arckeywords "kw1;kw2;..."
-arcnotes "notes (\n?)"
-arcargs "uf80c args for subjob"
-arcsep char  (separator for category segments - default is |)
-arcdtm yyyymmddhhmmss
-arcsubdtm yyyymmddhhmmss
```

Using the Web Browser Interface

The web browser interface is used to browse, search, and view archives and associated images. The setup of the browser interface is simple; you can use the internal UnForm HTTP server, or use an external web server, such as Apache or Microsoft IIS, operating on a system that has access to the UnForm server. A CGI script, uf80a.pl or uf80a.exe, is placed in an appropriate scriptable location, or given an appropriate scriptable name, and users simply need to point their web browser to the address, such as http://mycompany.com/cgi-bin/uf80a.pl.

To access the internal HTTP server, simply use the URL http://*server*:*port*/arc. The *server* is the name or IP address of the UnForm server, and the *port* is the listening port configured in the [httpd] section of uf80d.ini. An example might be: http://myserver:27282/arc.

The user is presented a login form first. The first time it is used, an administrator can login with the name “admin” and password “admin”, and can then add additional users and change the admin password using the browser.

The Browse feature provides drill down capabilities through the library, document type, document ID structure, and optionally by date or category indexes.

The Search feature provides cross-library searching for generic text patterns (Simple form interface) or by specific field attributes or ranges (Advanced form interface). Selected archives can be viewed, exported, transferred, or copied.

Administrative features, such as user login management and library security set up, are available when an administrative user logs in.

A session cookie is used by the web browser interface. If cookies are not enabled in the web browser, then the browser interface will not function correctly. The lifetime of a session is controlled by the sesage=*hours* value in the [archive] section of uf80d.ini. If set to 0, a login is required each time the user starts their web browser. If set to some other value, then sessions last the specified number of hours before a new login is required.

See the **Web Script Installation** chapter for more details about installing and configuring the CGI script. The interface itself provides on-line help for a more detailed description of how to use the browser interface.

Direct Browser Access to Documents

It is possible to view documents and document images directly, bypassing the full user interface, by using one of the following URL structures:

http://*server*/*path*/*script*?a=vw&lb=*library*&doctype=*doctype*&docid=*docid*

http://*server*/*path*/*script*?a=vw&lb=*library*&doctype=*doctype*&docid=*docid*&subid=*subid*

The first form will load a document-level view page, which includes links to images. The second will load an image directly. In each case, the server and script path must point to the UnForm web interface script, and the library, document type, document ID, and sub ID values must be URL-encoded (i.e. spaces are replaced with "+" characters, and certain other characters are converted to %*hex*. Information about URL-encoding can be found in any HTML guide.

If the *subid* specified is simply "@", as in "…&subid=@", then if the document only has an @UnForm subid (ignoring the @text subid), that PDF document will be viewed. If there is no @UnForm subid or there are additional images, such as scanned documents, then the document view page will be shown.

If the browser interface session has expired, then a login screen will be presented before the document is shown. It may be preferable to configure sessions to last longer, several hours or a day or more, to avoid this requirement.

Alternatively, it is also possible to use one of the UnForm clients, in conjunction with the –p winpvw option, to retrieve and view a document image from an archive. This may be preferable as login information can be supplied from the command line or saved information from prior executions.

Customizing the Web Interface

The most common customization is to modify the main title and logo image. This is done by modifying the logo= and title= settings in the [archive] section of uf80d.ini.

For further customization, the browser interface can be customized to support preferred colors and layout, and even to translate to different languages. The basic web interface is stored in a directory called web/en-us in the UnForm server directory. Within this directory are many .html files that act as templates, a messages.txt file that contains text messages and other items, and two style sheet files, default.css and custom.css. Several templates, including master.html and popup.html, provide the overall layout of the web interface.

The files in web/en-us are all subject to overwriting when UnForm is updated, so to provide a custom interface, you should copy any files you want to customize into a different directory under the web/* structure.

In addition, each custom web/* directory requires its own messages.txt file, even if it is a copy of the one provided in en-us.

Once this is done, you can modify the webdirs= line in the [archive] section of uf80d.ini. This line contains a list of directories that the user login screen lists as “Language” choices. There can be many directories, or just one. Whichever is selected by the user, that directory becomes the first location searched for web files, with the en-us directory searched after for any files not found. As the en-us directory is also searched, there is no need to copy all files to the custom directory – just those you want to modify.

The name=*value* line in the web directory’s messages.txt file is used as a description in the login screen.

Note that the first webdirs entry can contain a different login.html template, and this becomes the form shown when users first login.

While it is possible to modify the cosmetic appearance of the browser interface, it is not possible to modify the underlying structure or navigation, so be sure that if you modify templates all bracketed tags and links are maintained.

Note that if the publisher makes changes to the master en-us structures, any custom templates will have to be updated to reflect any new options, tags, or other critical elements related to processing.

Caution: Any modifications performed to the templates should only be performed by experienced and knowledgeable HTML and CSS programmers. SDSI cannot warrant the performance of customized templates.

Using the UnForm Client

The UnForm clients, which include the Perl-based Unix client (uf80c), and the Windows client (uf80c.exe), can all perform document management functions via command line options. In many cases, a login is required. This can be supplied via the -arclogin option in the form “*userid*/*password*”. In addition, the clients can prompt for login information by supplying the special syntax **–arclogin ask**, and/or this information can be stored.

If login information is not supplied:

- The Unix client will look in the files $HOME/.ufarc or /etc/.ufarc for two lines, login=*userid* and pswd=*password*. These are stored in clear text, so the only effective security for this is to use user-specific .ufarc files (in $HOME) and make sure they are readable only by the user.

- The Windows graphical client will prompt in a window for login information, and can optionally save this information so that it will not continue to prompt. To reset this information, use the -arclogin ask option to force a new login window. The stored, encrypted information can be removed from the uf80c.ini file in the local Windows directory to force a new login.

Triggering Archiving of UnForm Jobs

In addition to using archive commands in rule sets, you can use command line options to establish default values and to trigger archiving even in cases there rule sets do not contain archive commands, or if no rule set is invoked and jobs are passed through to output.

At a minimum, you should specify a library, using the –arclib option. Other options that set archive properties can be used, but care needs to be taken as these are generic options that will apply to all documents that are not archived via specific archive commands.

-arclib "libpath"
-arcdoctype "doctype"
-arcdocid "docid" (if not supplied, an auto-generated number is created)
-arcsubid "subdoc ID"
-arcsubtitle "subdoc title"
-arctitle "title"
-arccats "cat1.1|cat1.2|cat1.3;cat2;cat3.1|cat3.2;..."
-arckeywords "kw1;kw2;..."
-arcnotes "notes" (use \n character sequence to embed line breaks)
-arcentityid “entityid”
-arcargs "uf80c args for subjob"
-arcsep char (separator for category segments - default is |)
-arcdtm yyyymmddhhmmss (normally automatically calculated)
-arcsubdtm yyyymmddhhmmss (normally automatically calculated)

The keywords setting can be a semi-colon delimited list, or a number indicating how many keywords to generate from content (-1 or "all" for all). The default number of keywords is found in uf80d.ini. When keywords are scanned, there is a file (ufnonwrd.txt by default) that contains words and patterns to ignore.

Use \; or \| (or \<sep char>) to embed delimiters in keywords or categories. Use \\ to embed a backslash.

Note ; and | characters (and spaces) must be protected from the shell, so keywords and categories in particular should be quoted, as well as any argument with spaces in it.

-arcsubid "subid*" will sequence the key to prevent overwrites. Using "subid*" will force "subid*".

Adding External Documents

To add external documents to a library, bypassing any UnForm processing of the input, use the –arcput command line option, in conjunction with the –i option to name the file to add. Note this differs from archiving of UnForm jobs, as the input file is not processed through a rule set, but rather written directly into the archive. In addition, further options may (and probably should) be used:

```
-arclib "libpath"
-arcdoctype "doctype"
-arcdocid "docid"  (if not supplied, an auto-generated number is created)
-arcsubid "subdoc ID"
-arcsubtitle "subdoc title"
-arctitle "title"
-arccats "cat1.1|cat1.2|cat1.3;cat2;cat3.1|cat3.2;..."
-arckeywords "kw1;kw2;..."
-arcnotes "notes"  (use \n character sequence to embed line breaks)
-arcentityid “entityid”
-arcargs "uf80c args for subjob"
-arcsep char  (separator for category segments - default is |)
-arcdtm yyyymmddhhmmss  (normally automatically calculated)
-arcsubdtm yyyymmddhhmmss  (normally automatically calculated)
-arclogin "userid/pswd" | ask
```

The keywords setting can be a semi-colon delimited list, or a number indicating how many keywords to generate from content (-1 or "all" for all). The default number of keywords is found in uf80d.ini. When keywords are scanned, there is a file (ufnonwrd.txt by default) that contains words and patterns to ignore.

Use \; or \| (or \<sep char>) to embed delimiters in keywords or categories. Use \\ to embed a backslash.

Note ; and | characters (and spaces) must be protected from the shell, so keywords and categories in particular should be quoted, as well as any argument with spaces in it.

-arcsubid "subid*" will sequence the key to prevent overwrites. Using "subid*" will force "subid*".

Write access to a library is required to add a document to it.

Document Retrieval

The command line can be used to extract a document image from a library. To do so, you must supply the –argget command line option, along with identifying options to indicate the library, document type, document ID, and image sub ID. The –o option is used to specify where the document should be placed, typically with a “client:” prefix, like –o client:/tmp/myfile.pdf.

```
-arcget
-o filename
-arclib "libpath"
-arcdoctype "doctype"
```

-arcdocid "docid"
-arcsubid "subdoc ID"
-arclogin "userid/pswd" | ask

Read access to the library is required.

When used in conjunction with the –p winpvw driver, a local view of the document is presented on the workstation running the UnForm client.

When used in conjunction with –arcget, the –arcsubid option supports two special suffixes, “-<” and “->”, to return the first or last sub ID that matches the sub ID string. For example, a value of “@UnForm->” would return the last sequential @UnForm sub ID in an auto-sequenced library. Escape the suffix as “\->” to look for the literal value.

Document Deletion

The command line can be used to delete a document image from a library. To do so, you must supply the –arcdel command line option, along with identifying options to indicate the library, document type, document ID, and optionally an image sub ID.

-arcdel
-arclib "libpath"
-arcdoctype "doctype"
-arcdocid "docid"
-arcsubid "subdoc ID"
-arclogin "userid/pswd" | ask

If subid is supplied, just that subid is deleted. If no subid is supplied, the full document record is deleted, including category indexes and all subid images.

Delete access to the library is required.

Document Listings

To list libraries, use the –arclistlibs command line option:

-arclistlibs
-arclistfmt tab|csv|pipe
-o “output file”

To list document types in a library, use the –arclisttypes option:

-arclisttypes
-arclib “library”
-o “output file”
-arclogin “userid/pswd” | ask

To list documents in a library, use the following options:

```
-arclist
-arclib “libname”
-arclistfmt tab|csv|pipe|html|xml|xmlf
-arcorder id|date|title|category
-arcstart "starting point in order specified"
-arcend "ending point in order specified”
-arcfilter "filter string"
-arcsep char (used to parse start/end values for category segments)
-arclogin "userid/pswd" | ask
```

You must have read access to a library to list documents in it.

The columns listed include document type, document ID, date, time, title, the document storage file path, and entity ID. If you specify the category order, then the category’s segments are added as a single column. Otherwise, no category information is shown. If the list format is html or xml, then notes, keywords, links, and categories are added to the list.

The start and end range values relate to the order. For example, if the order is “id”, then the start and end ranges refer to document type and ID sequences. For segmented ranges, such as for type and ID values, or category segments when listing in category order, separate them with the –arcsep value (a pipe (|) by default). For example:

```
-arcorder id –arcstart “Invoices|000152”
```

Be sure to quote the range if it contains characters, such as pipes or spaces, which are significant to the operating system. For date ranges, enter dates in the structure yyyymmddhhmmss (as many characters are significant to your request). For example, -arcstart 200612011800 –arcend 200612020800 for the range from 6:00 PM on December 1, 2006 through 8:00AM on December 2, 2006.

To filter documents returned in the list, use the –arcfilter option. The filter can be a simple word or phrase, a simple wildcard containing * and/or ? characters such as “Acme*”, or a regular expression prefixed with a tilde, such as “~[0-9][0-9]\.[0-9][0-9]”. Filters are applied to a concatenation of the document type, document ID, date, time, and title values, and in the case of category order, the categories are filtered as well. If the list format is html or xml, then the notes, keywords, and categories are added to the list. Keywords and categories, though not category segments, are delimited with spaces rather than semicolons when filtered. If the list format is xmlf, then a base64-encoded version of the image file is also added to the xml document.

To list documents and their associated image information, use the -arclistdocs command line option rather than the -arclist option. The same options as shown above for –arclist apply.

When used in conjunction with the –p winpvw driver, a local view of the listing is presented on the workstation running the UnForm client.

Searching for Documents

To search a library, use the –arcsearch command line option:

-arcsearch
-arclib "*library*"
-arclistfmt tab|csv|pipe|html|xml|xmlf
-o "*output file*"
-arclogin "userid/pswd" | ask

Refine the search by adding any appropriate arguments from this list: -arcdoctype, -arcdocid, -arctitle, -arcentityid, -arcdate, -arcdtmupdated, -arckeywords, -arcnotes, -arccats, -arclinks, -arctext, or –arcsubid. Each of these arguments can be a wildcard (**value** or *value**), an exact value "12345", a range "12/1/2007-12/31/2007", or a regular expression ("~[0-9][A-Z]"). You can use "not" to look for archives that do not match a criteria, and "and" or "or" to search for multiple values or alternate values.

Searches are optimized when possible. The best optimizations are document IDs, entity IDs, small date ranges, and multi-level categories.

Note that document types and document IDs are case-sensitive when optimized.

Testing Existence of Documents

To test the existence of a library, document type, document ID, or image sub ID:

-arcexists
-arclib "*library*"
-arcdoctype "*type*"
-arcdocid "*docID*"
-arcsubid "*subid*"
-o "[client:]*outputfile*"

No login is required. You can test for just the library, a library and document type, a library and document type and document ID, or all four elements. The system prints a 0 or 1 to the output device, plus a CRLF (0=not found, 1=found).

Importing Documents from sdStor

UnForm can import images from an sdStor library. It performs this by extracting the text documents from the library and running them as UnForm jobs. The import is performed for a single library, specified using the –arcimport option, which also allows specification of the sdStor login and password:

-arcimport "sdstor_libname;login/pswd"

The default UnForm library will match the path used for the sdStor library. To override this default, add: -arclib "libname".

A few additional command line options are added automatically in order to:

- Retain the original date and time of the sdStor document
- Retain the title and keywords from sdStor
- Add an additional keyword "sdstor *sdStorID*"
- Add a category "sdstor|*stStorID*"

The keyword and category additions are provided to help link documents in sdStor with documents added via the import to an UnForm library.

For enhanced processing, specify a rule file using the –f *rulefile* option, and use **archive** commands in the rule file to provide the document settings desired. When using a rule file, be sure to not specify a date and time, so the command line options that capture the date and time from sdStor will not be overridden.

The import is processed by extracting all the documents from the sdStor library and placing them in the rpq directory (the direct TCP/IP printing queue), where they are automatically processed sequentially. As each document is added to the queue, a log line is printed to the command line's output, so it is recommended that a –o option be used to send log output to a server file (don't use the client: prefix on the output file), as imports can be time consuming. The amount of time spent extracting to the queue, and the amount of time it takes for the queue to be processed, depends on the number and size of documents in the sdStor library.

UnForm Image Manager

An optional companion product to UnForm archiving is the Image Manager client. This program can be installed on any Windows workstation on the network where the UnForm server is running. This tool is designed to obtain images from the Windows file system or any scanner available to the Windows system where it is installed. Those images can be identified automatically or manually, then uploaded to an archive library.

Typically such images are linked by document type, document ID, and a unique sub ID to other images in a library. For example, a signed delivery document could be identified by the customer and order number, and be stored as an image version of the order.

The Scanning Workstation supports OCR and barcode recognition, and VBScript-based job definitions, to help automate the process of image identification and management when dealing with known formats. This is particularly useful when scanning images generated internally by UnForm.

The Scanning Workstation includes a comprehensive help file that contains more information.

Functions Related To Archiving

When processing a job, UnForm can not only add documents to an archive, but it can also extract documents from a library for use in processing. For example, a statement job could be designed to extract a list of invoice PDF images and attach them to the statement using the **images** command.

To retrieve a document during an UnForm job, use this function in a code block:

getarc(*library$,doctype$,docid$,subid$,filename$*[,*errmsg$*])

If filename$="", it will return a temporary file containing the document. This temporary file will be erased at the end of the job. If a filename is supplied, that file will be created and will not be erased at the end of the job.

If errmsg$ is present, it will return any error if the document can't be found or if another unexpected error occurs.

To convert a PDF file into an image, using Ghostscript, use this function:

pdftoimage(*fromfile$,tofile$,format$* [,*resolution* [,*errmsg$]*])

This function will invoke Ghostscript, either on the UnForm server or on the UnForm Support Server, to convert the PDF file in fromfile$ to an image file in tofile$. The format of the output will is named in format$, and it must match one of the [driver] names found in uf80d.ini. The image is created at the dpi specified in resolution, or 300 dpi if not.

If tofile$ is null, it will be returned with a temporary file name that will be erased when the job is complete.

Any error message is returned in errmsg$.

Ghostscript must be configured in the [drivers] section of the uf80d.ini file, or an UnForm Support Server with Ghostscript configured must be available. Note that the **images** command automatically performs this step function if it encounters a PDF file name.

To test if a library exists, use the libexists() function:

libexists(*lib$*) returns 0 if library lib$ doesn't exist, or 1 if it does.

To obtain a list of image subids associated with a document, use the getsubids() function:

getsubids(*lib$,doctype$,docid$*[,*dlm$*])

Returns a list of document sub IDs, such as the @text and @UnForm subids automatically generated by the archive command. The list is returned as a delimited string, with the default delimiter being a semicolon. If the delimiter occurs in any subid, it is escaped with a \. The returned value may be used by the parse functions.

When a series of document images should be attached to a page produced by an UnForm job, you can extract the desired documents to work files, using the getarc() function, and append them to the page, optionally tiled, using the **images** command.

To determine if a document or sub document exists, use these functions, which return 0 if the specified entity does not exists, or 1 if it does:

> docidexists(*lib$*,*doctype$*,*docid$*)
> subidexists(*lib$*,*doctype$*,*docid$*,*subid$*)

Building Demo Archive Data

The UnForm server includes a Unix shell script called **arcdemo.sh** in the samples sub-directory. The shell script can be executed to initialize and build demo archive data based on the arcdemo.rul samples. Windows installations can produce the same list by choosing the Configure button and clicking the Other tab in the Server Manager.

Transferring Archives to A New System

The archive libraries are subdirectory structures that are binary compatible between operating systems (see exception, noted below), so it is possible to move them between Windows, Unix, and Linux servers if necessary. In addition to the library directories, there are several files in the UnForm server directory that reference the full paths of the various library directories, or are used for library security. These files are:

ufarcacc.dat	User-library access rights
ufarcgac.dat	Group-library access rights
ufarcgrp.dat	Groups table
ufarclib.dat	Libraries, identified by their full path
ufarcusr.dat	Users table

If the libraries are moved to the same path names on the new system, then all that is required is to copy these files to the new UnForm server directory. Note that simple library names are converted to use the full path of the UnForm server directory, plus the “arc” subdirectory, so the UnForm server should also be installed in the same path on the new system to take advantage of this capability.

If the libraries are to be moved to a new path on the new system, then the library, user-library access rights, and group-library access rights tables will need to be created using the browser administrator interface. In addition, any rule file archive commands and external scripts that reference library paths will also need to be updated before executed. The users table and groups table can be copied “as is”, since they do not reference library path names.

Note that a few operating systems do not support zlib compression. Libraries created on a system with zlib support are not compatible with systems that do not support it.

MIGRATING ARCHIVING FROM UNFORM 7.X TO UNFORM 8.0

UnForm 8.0 and 7.1 share the same archive library format, so it is possible for libraries created under 7.1 to be left in place, or copied "as is" to new library locations for 8.0 to use. In fact, it is safe to let both 7.1 and 8.0 maintain the libraries at the same time during a migration period. If libraries are moved to new paths, 8.0 adds a new library definition feature, "aliases", which allows a library path to be recognized as multiple names. This feature is useful when there are external links to libraries that can't be updated to utilize a new location.

Note that libraries from UnForm 7.0 are not 100% compatible with 8.0 library structures, so can't be shared between an active 7.0 and 8.0 server. Archive records updated by 8.0 will not be compatible with 7.0. Therefore, to migrate from 7.0 to 8.0, you must make copies of all libraries and ensure that 8.0 uses the new library paths.

Migrate to 8.0 on a new system

To migrate UnForm 7.x data to a new UnForm 8.0 installation on a new server, first install 8.0 on the new system, using the same directory as 7.x on the old system. Next, copy all library directories (including those in the default “arc” subdirectory) to the same paths on the new system.

Then copy these files to the UnForm server directory:

- ufarcacc.dat – library/user access table
- ufarcgrp.dat – groups table
- ufarcgrc.dat – library/group access table
- ufarclib.dat – library full paths and properties
- ufarcusr.dat – user table

If you wish to use new paths for any libraries, you may do so, but will need to use the browser interface Library maintenance function to change library path names before any archiving activity or browser access to the libraries takes place. Any library location changes require that the library table be updated to reflect the new physical library locations.

If using, install the uf80a.pl (Unix) or uf80a.exe (Windows) to the cgi scripting directory on your web server, and create a uf80a.ini file in the same directory with a server=*name-or-IP* configuration line to direct the web server to the new UnForm server (or begin using a local web server on the new system).

In order to continue to print from the old system to the new one, install an UnForm 8.0 client on the old system, and add a –server *name-or-IP* option to the uf80c command lines to submit print jobs to the new server.

Migrate to 8.0 on the same system

To migrate to 8.0 on the same server as the 7.x installation, install the 8.0 server into a new directory, then copy any libraries in the former 7.x "arc" directory to the new 8.0 "arc" directory. Other libraries can be left in place or moved if desired (though 7.0 libraries should be copied rather than left in place unless 7.0 is no longer to be used). Any library location changes require that the library table be updated to reflect the new physical library locations.

Take steps to ensure no UnForm printing will take place to the 8.0 server, then copy these files to the UnForm server directory:

- ufarcacc.dat – library/user access table
- ufarcgrp.dat – groups table
- ufarcgrc.dat – library/group access table
- ufarclib.dat – library full paths and properties
- ufarcusr.dat – user table

Use the browser interface Library page to modify the library paths to each of your new library locations. As you do so, the custom user and group access records will be updated with the new library name.

Be sure to update any rule files that have full paths to library names in archive commands, so printing archives will update the new library locations.

At this point, you can begin submitting print jobs to the UnForm 8.0 server.

WINDOWS SUPPORT SERVER

The Windows Support Server is a no-charge companion product that can be installed on any Windows 2000 or higher computer on the network where the UnForm server runs. If the UnForm server is running on Windows, then the Support Server is already installed and can be enabled through server configuration. If the UnForm server is not running on Windows, you can install the Support Server separately and configure these lines in the [defaults] section of uf80d.ini:

sshost=*hostname or IP address of Support Server*
ssport=*listening port number*

In addition, the sshost() code block command can be used to set the support server machine and port during job processing.

The following features are supported:

- **Image scaling and conversion**
  The UnForm server can utilize a local copy of Image Magick or Image Alchemy to perform image scaling and conversion, but these products are not always readily available for some Unix operating systems. In addition, the **image** command supports two options, gamma *n* and rotate *n*, which the Support Server honors. This feature is used automatically by UnForm whenever an image conversion or scale is required, if Magick or Alchemy is not configured for use at the server.

- **GhostScript-based Image Output**
  The UnForm server can utilize a local copy of Ghostscript to produce image output and to convert PDF files to other formats. However, the latest versions of Ghostscript are not readily available on all operating systems. By installing a Windows version of Ghostscript on the support server, the UnForm server can rely on it to perform the conversions. This feature is used automatically by UnForm whenever a PDF-to-image conversion required, if GhostScript is not configured for use at the server.

- **Database Access**
  The support server can be configured to access data base sources via ODBC or more recent database access technologies. UnForm rule files can connect to these data sources and retrieve data for use in UnForm jobs.

  Data sources are configured using the Support Server configuration window. UnForm jobs can then use the dbconnect() and dbexecute() code block commands.

- **Microsoft Fax**
  The Microsoft fax server, a free product available for Windows 2000 and up (and pre-installed on Windows XP and up), can be easily set up on the support server or another Windows server

on the network. The support server can then use the Microsoft fax client to send faxes on behalf of UnForm jobs.

UnForm jobs can use the msfax() code block command as soon as Microsoft Faxing is configured.

- **Extended Barcode Functionality**
  Support for additional barcode options (for pcl, pdf, and postscript output) is offered with the Windows Support Server. Two new 1D symbologies plus four 2D symbologies are supported, plus rotation and human-readable text lines and some other options. See the barcode command for more details.

Configuration note: in order to fax PDF documents using the Support Server, you must install and configure Ghostscript on that server. The Support Server can then convert PDF files to tif for faxing. The reason for this is that Acrobat doesn't support the Windows shell's "printto" action, which Microsoft Fax uses to convert documents to faxable tif format.

The following table describes the various code block functions that are supported when the Support Server is available.

sshost(SERVER$,PORT)	Sets the support server hostname and port. Default values are defined in the uf80d.ini file in the sshost and ssport settings. This command allows for dynamic changing to a different server.
dbconnect(NAME$, TIMEOUT, EMSG$)	Connects to the database source identified by name$. The support server configuration is used to define the names and associate them with data source connection strings. Typically done in a prejob code block.
dbexecute(NAME$, CMD$, TIMEOUT, FDELIM$, RDELIM$, RESPONSE$, EMSG$)	Executes the SQL command cmd$ and returns zero or more result rows in response$. Columns are delimited by fdelim$ (tab - chr(9) - by default). Rows are delimited by rdelim$ (CR-LF - chr(13)+chr(10) - by default).
msfax(FILENAME$, FAXNUM$, TAGS$ [, EMSG$])	Faxes filename$, normally an UnForm-generated PDF file, to the fax number specified in faxnum$. Numerous supported tags can be specified in tags$, in the format tag1=value;tag2=value,... The format of faxnum$ can be a simple phone number, or multiple numbers separated by semicolons, or tags in the format: name1=fax1; name2=fax2, ...

Quote the entire tag if it contains semicolons: "Smith; Cline; Robert=9,1-555-555-5555". This will involve use of quote characters in the expression, using chr(34) or 22. For example:

Faxnum$=chr(34) + name$ + “=” + faxno$+chr(34)

Note that fax numbers may need to be complete, using for example "9,1" as a prefix for an outside line, a pause, and a leading 1 before the area code, depending on fax server use of dialing rules.

Tags supported are:

Cover	Standard coversheet name based on the fax server, such as cover=generic.
Localcover personalcover	Personal or fax client-side cover sheet.
Subject	Subject for cover sheet.
Note or Notes	Notes for cover sheet. Use \n for hard line breaks.
Time	A human-readable date and time to send the fax, if not immediately.
Receipt Attachfax	An email address to send fax result reporting to. Note that you must be using a Server version of Microsoft Fax with Microsoft Exchange and enable SMTP receipt delivery for this to work. If a receipt is specified, you can additionally use the attachfax option to have the receipt email include the fax image.
Alert	This tag’s presence requests that the fax client issue a message box regarding the fax disposition.
Server	Set the Microsoft Fax Server computer name, if the server is not running on the same system as the UnForm support server.
ToName	If a single fax number is supplied, this tag is an alternate way to specify the recipient name.
FromName	Alternative tag to set the sender name.
FromCompany	Alternative tag to set the sender company name.

	Other sender tag names that may be used by a cover page: name title company department title homephone officephone faxnumber email streetaddress city state zipcode country

DESKTOP DELIVERY AND FORMS

UnForm 8.0 provides two features that enable users on the network where the UnForm server runs to receive documents or fill out forms at the time UnForm jobs run. These features are provided via a high-performance HTTP server that is included as part of the UnForm server. This HTTP server is configured via the [httpd] section of the uf80d.ini file (the port can also be configured in the server manager when UnForm is installed on Windows). By default, the port is 27282. Once this server is running, users can use any web browser to connect to the server using one of these URLs:

http://*servername*:*port*	Presents a portal page with links to the delivery browser or monitor.
http://*servername*:*port*/dtbrowse.html	Presents the delivery browser, which displays documents waiting for the user, and monitors for form requests.
http://*servername*:*port*/dtmonitor.html	Presents the delivery monitor, which is a small browser window that provides a summary of documents waiting for the user, and monitors for form requests.

Optionally, you can supply a query string suffix "?ips=*uniqueid*" to the above URL structures. The *uniqueid* value can be something that will add uniqueness to the IP address that the browser monitors with, supporting Terminal Server environments where the same IP address would be used for all users. Typically there is an environment variable, such as %SESSIONNAME%, that identifies a particular session on the server. By passing this information to both the URL and UnForm job submissions, deliveries and forms can be sent to the correct user session.

When users connect to the server, they must login using an UnForm login. These logins are maintained using the web browser interface used to administer archive access, so user maintenance serves a dual purpose. This login is maintained using standard HTTP authentication, and the browser may or may not offer to remember the login.

Note that in addition to the desktop delivery addresses above, the browser interface for archiving and administration can be accessed as http://*servername*:*port*/arc.

Desktop Delivery

The delivery of documents is performed by the dtdel() code block function. When this function is executed, a copy of the file to deliver is stored securely on the server. The desktop browser and monitor poll the server for new documents to display, and when new documents are available, they are listed as available or are immediately displayed.

Documents are not stored indefinitely. The system purges documents based on the dtage and dtviewage parameters in the [defaults] section of uf80d.ini. These values specify the number of days unviewed documents, and viewed documents, are maintained, respectively.

The dtdel function has the following syntax:

dtdel(*filename$*,*title$*,*userid$*,*ip$*[,*style*[,*errmsg$*]])

The filename$ argument can be either a file that resides on the UnForm server or a message in the format of "msg:*message text*". It will typically be a PDF file that was generated using a jobexec() command, or it could be the job's output file, delivered in a postdevice code block. If it is a message, the text is merged with the message template "http/files/msg.html", and it is always treated as a popup (style=1). As the popup is based on an HTML template, you can embed HTML code in the *message text*. You can also use "\n" as a synonym for "<br>" as a convenience.

The title$ argument is a document title that will display in the delivery browser.

The userid$ and/or ip$ arguments are used to identify which user should receive the document. The preferred method is to specify a userid$ value. If no user ID is specified, then an IP address can be used. Only a browser logged in as the specified user, and/or connected from that IP address will be notified of the delivery and will be able to access it. The IP address reported in uf.clientip$ is typically the IP address of the computer that submitted the UnForm job, though in some circumstances it will be the address of a server computer rather than a user computer.

As IP addresses can change if DHCP is used or static IPs are re-assigned, or may not be available if the user is accessing the server via a router with NAT translation, care must be taken when using an IP address. They are suitable in local networks for popup styles of delivery, assuming that purge times (dtage and dtviewage) settings are short.

A suffix can be appended to the ip$ value. This suffix must match the value used in an ips=*uniqueid* query string in the browser monitor launch. For example: uf.clientip$+clientenv("SESSIONNAME").

The style argument can be 0 or 1. A value of 0 indicates the monitor and browser windows will display the presence of the delivery. A value of 1 indicates that the browser will immediately display the document as well. If the style argument is not supplied, 0 is assumed.

The errmsg$ variable will return an error message if an error occurs while storing the document. It will return null ("") if no error occurs.

Desktop Forms

Desktop form support is configured the same way as desktop delivery. The same HTTP server and clients are used. Unlike desktop delivery, which will store documents for when the user logs in, it is critical that the user be connected when the form is to be presented.

The form is launched with the dtform() code block function. The user specified is notified that a form is ready to be presented, and he or she can accept or cancel the form. If the user neither accepts nor cancels the request within a specified amount of time, the request times out.

Data is sent to the form, and returned from the form, using a URL-encoded data string. There are several functions provided to manage this string.

The form itself is an HTML form document, stored in the http/files directory under the UnForm server. These forms should have a hidden field called cancel with a value of 0 for a regular submission, or 1 for a cancelled form. Other than that, any standard HTML form widgets can be used, including text boxes, text areas, radio buttons, checkboxes, and selection lists.

To provide default values for form fields, specify the named value(s) in the data string argument of the dtform command, and include ~*name*~ tags in the HTML document. For example an input field for a To address might look like this:

```
<input type="text" size="30" name="to" value="~to~">
```

If the data string supplied to the dtform function contains to=someone@somewhere.com, then that email address will be presented as the default value when the form is displayed.

The dtform function has the following syntax:

dtform(*formname$*,*title$*,*userid$*,*ip$*,*datastr$*,*response*[,*timeout*[,*errmsg$*]])

The formname$ value is the base name of the html form file found in http/files. For example, “emailform.html” could be used to use the sample email form provided with UnForm.

The title$ value is what is shown to the user when notified that the form needs to be displayed.

The userid$ and/or ip$ arguments are used to identify which user should receive the document. The preferred method is to specify a userid$ value. If no user ID is specified, then an IP address can be used. Only a browser logged in as the specified user, and/or connected from that IP address will be notified of the delivery and will be able to access it. The IP address reported in uf.clientip$ is typically the IP address of the computer that submitted the UnForm job, though in some circumstances it will be the address of a server computer rather than a user computer.

A suffix can be appended to the ip$ value. This suffix must match the value used in an ips=*uniqueid* query string in the browser monitor launch. For example: uf.clientip$+clientenv(“SESSIONNAME”).

The datastr$ is a URL-encoded string with form field values defined as name=value pairs as in normal web programming. It must be a string variable in order to receive form values back, which can then be decoded using the urlgetfld() function. If the string contains values when dtform is executed, those values are used in the form, wherever a ~*name*~ tag is found. To create the string with URL-encoded name=value pairs, use the urlsetfld() function.

The response variable returns one of these codes:

- 0 indicating the form was submitted
- 1 indicating the form was cancelled

- 2 indicating the form request timed out
- 3 indicating the user refused the form

Any non-0 responses are logged in the server log file. Rule sets that use the dtdel() function should query and react to non-0 responses as appropriate.

The timeout value is the number of seconds the user has to respond to the form request. Once the user accepts the form, they may take as long as needed to complete the form. However, the job will be halted waiting for the form submission, so users must understand that forms they accept should be submitted as soon as possible. If the user doesn't accept the form within the specified number of seconds, a timeout response will be provided. The default timeout value is 30 seconds.

If a timeout of -1 is specified, then the form request step is skipped, and the form is displayed automatically. If no monitor is running or the user does not respond to the form, the job will be hung, so do not use this option unless you know the monitor is active and the user is available. This step might be used as an immediate follow up to a previous form that the user did respond to.

If an unexpected error occurs, it will be returned in errmsg$, if provided in the function arguments.

URL-encoded strings are comprised of name-value pairs with special character encoding. Use the following functions to create or parse the URL-encoded data string:

urlgetfld(datastr$,name$)	Returns the value of the name$ field. The value is returned without URL encoding. mailto$=urlgetfld(datastring$,"to")
urlsetfld(datastr$,name$,value$)	Returns a URL-encoded string with the field name$ set to value$. The field is added if necessary. datastring$=urlsetfld(datastring$,"to",someone@somewhere.com)
urldelflds(datastr$,names$)	Returns the a URL-encoded string after removing the fields specified in name$ from the URL-encoded string datastr$. Multiple fields can be separated by commas. datastring$=urldelflds(datastring$,"to,from,subject,body")

urlgetnames$(datastr$)	Returns a list of field names in the data string.
	
	fldlist$=urlgetnames$(datastring$)
	count=parsec(fldlist$,",')

ADDRESS BOOKS

UnForm supports multiple address books to assist in delivery of documents to email or fax addresses. Address books can be created and maintained directly in the archive browser interface, or programmatically via rule set code blocks.

Address books utilize a concept of an Entity ID, which is an identifier for a particular entity, such as a vendor or customer, or any other unique contact that might be required for an application. In addition to an entity ID, an address record is identified by an optional document type. This allows a single address book to be utilized, for example, for customer addresses for sales, delivery, or ad hoc contact. Each combination of entity ID and document type can have a delivery address, either email or fax.

Specifically, the fields maintained in an address book record are:

- Entity ID
- Document Type
- Entity Name
- Contact Name
- Send To (email address or fax number)
- Combine

The combine field can be leveraged by the **deliver** command, which has the capability to combine multiple documents in a batch that are targeted to the same delivery address.

The browser interface provides address book maintenance features to users who are granted address book maintenance rights. This maintenance feature includes an ability to import and export an address book in a CSV format for easy maintenance using third-party tools or text editors. Many applications and report writers can produce CSV files, allowing the upload of address book information.

In addition, when address books are populated with entity ID's that match document entity ID's, email address suggestions are offered when viewing documents in the browser interface.

In addition to user interfaces, address books can be programmed within rule sets. There are two functions for simple read and write of address book records, getaddress() and putaddress(), plus the "addrbook" object, that provides greater flexibility. Using these facilities, you can create rule sets for importing and management of address book entries, as well as utilizing address books for delivery addresses for emailing, faxing, or the new **deliver** command, which offers batch handling and automated email and fax delivery capabilities.

DATABASE ACCESS

UnForm supports access to databases from rule set code, using one of two techniques.

Windows Support Server Access
One technique users the Windows Support Server to access data sources available on the machine where the support server runs. The Windows Support Server configuration window enables database connections to be configured and given a name, and two code block functions: dbconnect() and dbexecute() are provided to communicate with the named connection to return the results of a query. The syntax of these two functions is:

```
[success=]dbconnect(name$[,timeout[,errmsg$]])
[success=]dbexecute(name$, command$, timeout, fdelim$, rdelim$, response$ [,errmsg$])
```

Both functions return 1 if successful, 0 it not. This technique was available in starting with UnForm 7.0.

Server-based Access
The second technique, added in UnForm 8.0, supports access to database sources directly within the UnForm server. When using this method, you connect to a data source identified with a string construction, optionally supplying a user and password login, as well as other optional arguments. The sqlconnect() function provides the functionality, and returns a connection channel number.

Note that secure passwords can be configured in the browser interface and referenced in the sqlconnect() function, using the syntax "store:*ID*" rather than a plain text password.

After connecting, send SQL commands to the database channel using the sqlexecute() function. Access the data returned by the command, if any, using the sqlfetch() function, which can return one, many, or all rows from the query, in a delimited string.

When done with the data source, you can close the channel with the close(chan) command.

The syntax of the three functions is:

```
chan=sqlconnect(datasource$[,user$ ,pswd$ [,otheroptions$ [,errmsg$]]])
[success=]sqlexecute(chan,command$[,errmsg$[,result$[,fdelim$[,rdelim$]]]])
count=sqlfetch(chan,result$[,count [,errmsg$ [,fdelim$ [,rdelim$]]]])
```

There are four types of databases supported, though not all types are supported on all platforms. The "uf80c -v" command shows which database types are supported. The four types are ODBC, Oracle, DB2, and MySQL. Note that ODBC is supported on Unix/Linux, as well as Windows, if either the unixODBC or iODBC package is installed.

The syntax of the datasource$ argument identifies the database type and data source:

- odbc:*dsn* connects to the ODBC data source name (also called the DSN), as configured in the Windows ODBC administrator or in the unixODBC/iODBC configuration.
- oracle:*sid* connects to the Oracle System ID, using local Oracle client libraries.
- db2:*database* connects to the DB2 database specified.
- mysql:*database*[:*hostname*] connects to the MySQL database named, optionally on the host specified.

Most databases require a login and password in order to access a database. The user and password must be supplied in those cases.

Additional options that can be supplied in the otheroptions$ argument, as a semicolon-delimited list. Options include:

- access=read|write
- strip (if present, causes trailing spaces to be trimmed from fields)
- textmax=*val* (sets the maximum amount of text returned from a text field, default=4096)
- timeout=*seconds*

Once a connection channel has been created, you can then send SQL commands to the channel using the sqlexecute() function. That function can optionally fill a results variable with all the rows returned by the query, or you can use the sqlfetch() function to return rows one or many at a time.

Below is a simple example showing how to use the three functions:

```
prejob{
chan=sqlconnect("odbc:sampdb","userid","password")
if chan>0 then:
        e=sqlexecute(chan,"select member_id, last_name, first_name from member")
        if e>0 then:
                while sqlfetch(chan,row$)
                        row$=sub(row$,$09$,"|")
                        allrows$+=row$+$0a$
                wend
        end if
        close(chan)
end if
}

text 10,2,{allrows$}
```

DESKTOP CLIENT

The UnForm Desktop Client (DTC) client is an optional Windows application that provides streamlined access to UnForm document management facilities from a user's Windows desktop. DTC communicates with UnForm via the internal HTTP server. It downloads a specialized rule file that controls its processing. The primary purpose of DTC is as a monitor that watches for windows to appear and gain focus, present buttons related to those windows, and to submit data to the unform server to retrieve documents or perform other actions related to the unform server.

The configuration of DTC includes a user login and password, which is used to submit the data for rule set processing. Note this does not replicate to the user's browser, so separate session logins are required when a job launches a browser window.

The rule file contains rule sets, which are composed of three things: detection statements, button and panel commands, and a prejob code block that is executed when the user clicks a button.

When the rule file is loaded by DTC, it then begins monitoring for window focus changes on the desktop. When a window passes detection for a rule set, a small, user-sizable window is displayed with the defined buttons. When the user clicks a button, a job is run on the server, which executes the rule set and returns the value of cgiresponse$ to DTC, which then displays the response in one of several ways.

Note that since detection can be based on user rather than window, it is also possible to construct an interface that always displays regardless of what window currently has focus on the user's system.

Deployment

DTC can be installed directly from the server's "dtcinst" folder, by running the setup.exe or uf8dtc_setup.msi file. It can also be installed with a browser via the internal HTTP server portal, using the server's address and HTTP port, such as http://192.168.1.10:27282. Note this is one level higher than the normal /arc path to the server.

Once run, DTC will attempt to communicate with the server, but will need to be configured with a login and password, and an optional rule file. Right-click the "uf" DTC system tray icon to configure. A default rule file can be configured in the [dtc] section of uf80d.ini.

DTC Rule Sets

Desktop Client rule sets are similar to print job rule sets in many respects, but are limited in structure to a small number of commands. Other commands are ignored.

DTC rule sets require three things: detection logic, button definitions, and a prejob code block that creates responses to buttons that are pressed by the user and data that is submitted from DTC. Detection statements are used to specify which applications and/or window titles to monitor as focus changes between applications and windows on the user's workstation. As the active window or application changes, different rule sets become active and an associated DTC application window will appear. The contents of that window are controlled by other rule set commands: dtcbutton, dtchelpfile, and dtcpanel. These commands are use to construct and present a user interface associated with a window or application. The user can then copy, paste, or type data into the button text fields, and when a given button is pressed, the rule set is executed on the server, and the prejob code block is executed.

The syntax for the above mentioned rule set commands is as follows:

Detect

Detect 0, *item*, "[^][~]*match text*"

- 0,0 tests space separated exe name, window title, and UnForm login name
- 0,1 tests window title
- 0,2 tests exe name
- 0,3 tests UnForm user login
- 0,4 tests Windows domain\user

A prefix on the detection text can specify a case-insenstive match, regular expression match, or case-insensitive regular expression match:

- ^ case insensitive text match
- ~ case-sensitive regular expression
- ^~ case-insensitive regular expression

Each time window focus changes on the user's system, detection is processed for all rule sets. Any rule sets that pass detection cause presentation of that set's application popup window. It is possible for multiple windows to be displayed at the same time.

Title

Title "*window title*"

The title command can be used to specify the application popup window title. The title defaults to the rule set name.

DTCPanel

DTCPanel "*title*"

The dtcpanel command names a tab panel for subsequent button commands. When a form is submitted, panel titles become section headers for the cgi.data$ field, which is an INI file structured string. If no panel commands are present, a single un-tabbed panel is presented, and the cgi.data$ field contains a single section header, [*].

DTCHelpfile

DTCHelpFile "*filename*"

If present, the DTC window will offer a help toolbar button. The purpose is to allow integrator-generated help content associated with the rule set's DTC window. The filename should be an HTML file available on the UnForm server. The file is loaded into a browser control.

DTCButton

DTCButton "*title*" [,style clipboard|nbclipboard|text|nbtext|button|title|nbtitle] [,args "*job arguments*"] [,width *chars*] [,match "*regex*"] [,library "*name*"] [,doctype "*value*"] [,parsevalue ["*ruleset*"]]

The DTCbutton command is interpreted by DTC for presentation and action upon click. Note that expressions are not supported, only literal values.

Style
This optional argument defines the style of the button provided to the user. The default style is "clipboard".

Clipboard	Provides a text box and a small submit button, and monitors the clipboard for changes to place text in the text box.
Nbclipboard	Provides a text box, but no submit button. It monitors the clipboard for changes to place in the text box.
Text	Provides a text box and a small submit button. This is intended for simple user entry.
Nbtext	Provides a text box, but no submit button.
Button	Provides a submit button with the title as the button caption.
Title	Provides a text box and a small submit button, and monitors the window title for changes to place in the text box.
Nbtitle	Provides a text box, but no submit button. It monitors the window title for changes to place in the text box.

Args
Options passed to the to the UnForm job running the rule set. There are automatic rule file and rule set arguments (-f and –r, respectively) to ensure the rule set with the button configuration is the one executed when a submission takes place.

Width
This defines the width of the button, in nominal characters. Without a width, the title width is used. Use this to ensure consistent widths when multiple buttons are presented.

Match

The match option is honored by the clipboard and nbclipboard styles. The clipboard value must match the specified regular expression in order to be pasted into the text field.

Library, Doctype
These two options in tandem are honored by the clipboard and nbclipboard styles. The clipboard value must be a valid document ID within the library and doctype specified in order to be pasted into the text field.

ParseValue
If this option is present, the value from the clipboard or window title is sent to the server for parsing. The server will run the current rule set or the optionally specified one, and use the value returned from the server in cgiresponse$ in place of the clipboard or title value. The rule set receives cgi.button$, cgi.panel$, and cgi.parsevalue$ when the request is sent.

Code Block Response For Buttons

Once one of the buttons is pressed, the equivalent of a web form is submitted to the UnForm server, to run the same rule set that contains the detect and button commands (-f and –r are used, along with any args options and a hard-coded "-p pdf"). The following values are available in the cgi$ template:

cgi.button$	Returns button title, indicating which button was pressed.
cgi.selected$	Returns text of a text field associated with the button.
cgi.panel$	Returns the name of the active panel when the button was pressed, or "*" if no panels are defined.
cgi.data$	Returns all the panels and text fields in an INI file format. Panel titles are used as section headers, and text and clipboard field text boxes are returned in name=value format. If no panels are provided, a single section header [*] is supplied. This format allows multiple fields of data to be submitted and interpreted by the rule set, by using the getinival() function, passing cgi.data$ as the first argument. For example, name$=getinival(cgi.data$, "CustPanel", "Name") would set name$ to the value of the Name field in the CustPanel panel.
cgi.parsevalue$	Returns the value of the clipboard or window title in cases where the parsevalue option is specified. Note that if this value is present, DTC is submitting a request for reformatting of this data. It is not a result of the user pressing a DTC submit button. No cgi.selected$ or cgi.data$ is sent.

Use the prejob code block to interpret these items, and generate a cgiresponse$ string value. This value is returned to DTC for processing. DTC interprets the response in these ways:

- http: or https: prefix performs shell launch to display URL in the default browser
- error: message text (use \n for CRLF) displays an error style message box
- message: message text (use \n for CRLF) displays an information style message box
- other text renders in local browser control, and assumes HTML content

The webapi object can be used to generate http responses. Note these http values can also be used in <a href=> tags in a pure html response.

If no cgireponse$ is available, the print job result is returned instead, allowing artificial print jobs to execute (by generating text pages in a prejob code block, and allowing normal rule set processing to handle the job). The print job will be in PDF format.

Code Block Response For ParseValue Requests

If a clipboard or title button is configured with the parsevalue option, DTC will submit that value to the server for parsing.

The following values are available in the cgi$ template:

cgi.button$	Contains the button title, indicating which button contains the parsevalue option.
cgi.panel$	Contains the name of the active panel, or "*" if no panels are defined.
cgi.parsevalue$	Contains the value of the clipboard or window title in cases where the parsevalue option is specified.

Use the prejob code block to interpret these items, and generate a cgiresponse$ string value. This value is returned to DTC for processing. DTC interprets the response in these ways:

- error: message text (use \n for CRLF) displays an error style message box
- other text is used to replace the original value from the clipboard or window title

RULE FILES

Rule files are text files that contain descriptions of form enhancements. There can be any number of these enhancements, called *rule sets*, in a rule file. A header line composed of a unique name enclosed in square brackets indicates a new rule set. For example, an invoice form rule set would begin with the line **[Invoice]**, followed by lines indicating enhancements to the invoice output sent by the application. Without a rule set to work with, UnForm will not perform any enhancements. UnForm determines which rule set to work with based on either a command line option (-r), or **detect** commands contained in the rule set.

The enhancements that follow the [*form-name*] line are made up of commands and (usually) a list of parameters separated by commas. The available enhancements are described on the following pages.

Unless otherwise noted, all column and row specifications are 1-based (i.e. the first column is 1, rather than 0).

Commands that have parameters accept either a space or an equal sign between the keyword and the first parameter; **page 66** and **page=66** are equivalent.

If a command and its parameters require a large amount of text, it is possible to split a command across multiple lines by adding a backslash character at the end of a line to indicate the command continues on the next line. You can have as many continuation lines as necessary. UnForm removes leading spaces and tabs from continuation lines, so you can use indention to improve readability, as long as you remember to place any required spaces before the backslash on the initial line. For example:

```
text 1,30,"This line of text is continued \
          on this line.",12,cgtimes
```

Note that the UnForm Design Tool puts continuation lines back together, so this feature is useful only when using a text editor for rule file development.

The driver differences and support for different keywords is noted. Note, however, that when a command indicates all drivers, this doesn't necessarily indicate support by html. For the HTML driver, please refer to the HTML chapter.

Content-based Rule Sets

In addition to rule files, it is also possible to include a rule set in the content of a job, by beginning the job with a name in square brackets, like [ruleset]. If UnForm sees this line structure as the first line of a job, it then reads the input stream until it encounters a form-feed (ASCII 12, hex 0C), and then doesn't process the rule file at all. Instead, it uses the rule set provided for the job. The first character after the form-feed is treated as the start of the document, so take care that you don't have an extra line-feed that would throw off line numbers.

Using this technique, it is possible for applications such as report generators to enhance output programmatically.

ACROSS

Syntax

across *n* [,*gap*]

Description

This instructs UnForm to allocate virtual pages across the physical page, evenly spaced within the left and right margins. Use this feature for multi-up printing of standard reports, or for laser labels.

UnForm will automatically scale text (to as small as 4 point), boxes, and shading. It will not scale images, barcodes, or attachments. Also see the **down** command.

Across can be used inside an 'if copy' block, but is only compatible with non-collated copies. As a result, copy-specific **across** is only available in the laser driver, and only in conjunction with the **copies** command, not **pcopies**.

If the optional *gap* value is specified, it indicates the number of horizontal pixels between each virtual page. If it is not specified, the default is to use one *column* (as opposed to pixels).

See the 132x4 rule set in advanced.rul for an example of using the across and down commands.

Drivers: laser, pdf, ps

PostScript input not supported.

ANNOTATE, CANNOTATE

Syntax

1. annotate *col*|{*numexpr*}, *row*|{*numexpr*}, *cols*|{*numexpr*}, *rows*|{*numexpr*},"*msg* | *url*"|{*expr*}, [text|link|stamp] [,name *name*|{*expr*}] [,title *title*|{*expr*}] [,width *width*] [,color *colorname*] [,rgb *rrggbb*] [,opacity *opacity*] [,style *style*]

2. annotate "*text*|!=*text*|~*regexp*|!~*regexp*[@*left,top,right.bottom*]", *col*|{*numexpr*}, *row*|{*numexpr*}, *cols*|{*numexpr*}, *rows*|{*numexpr*},"*msg* | *url*"|{*expr*}, [text|link|stamp] [,name "*name*"|{*expr*}] [,title *title*|{*expr*}] [,width *width*] [,color *colorname*] [,rgb *rrggbb*] [,opacity *opacity*] [,style *style*]

Description

This PDF-only command adds an annotation element at the specified position and size. Three types are supported: text, link, and stamp. The default is text.

If **cannotate** is used, then *cols* and *rows* are interpreted to be the opposite corner of the region, and columns and rows are calculated by UnForm.

If syntax 2 is used, then the region is defined relative to any occurrence of the *text*, or of text that matches the regular expression *regexpr*. In these cases, there may be no affected regions, or several. c*olumn* and *row* are 0-based in these formats. The search for *text* or *regexpr* can be limited to a region on the page by adding a suffix in the format '*@left,top,right,bottom*'. To use a literal "@" character in *text* or *regexpr,* it is necessary to specify "\@".

If the syntax "!=*text*" or "!~*regexpr*" is used, then the search is for positions NOT equal to the text or NOT matching the regular expression. When using the NOT syntax, only one search is performed per line in the search region.

Text annotations display as an icon, which if clicked will open a text window displaying the message with the optional title in the title bar. The border can be controlled by the style, color, and width options. The icon can be set with the case-sensitive name option: Comment, Help, Insert, Key, NewParagraph, Note, or Paragraph. The default icon is Note. To ensure the case is maintained, the *name* should be enclosed in quotes.

Link annotations perform an action when clicked. The action is defined in the message. It can be a URL, such as http://abc.com or mailto:sales@abc.com, or it can be a Javascript action, entered as javascript:*script code*. Note the **javascript** command can be used to add functions at a document level which can be used in annotation actions.

Stamp annotations place a stamp rather than an icon on the form. When clicked, the message is displayed as in a text annotation. The stamp image shown is identified by the case-sensitive name value, which can be one of these:

- Approved
- Experimental
- NotApproved
- AsIs
- Expired
- NotForPublicRelease
- Confidential
- Final
- Sold
- Departmental
- ForComment
- TopSecret
- Draft
- ForPublicRelease

Style and Width values apply to the annotation border. The style can be S (solid), D (dashed), B (beveled), I (inset), or U (underline). Width is expressed in pixels at the current dpi.

Examples

annotate 70.5,64,10,2.25,"http://acme.com/docs/terms.htm",link,style U, title "Click to view our Terms and Conditions"

annotate "TOTAL:",0,1,15,1.5,"Please contact our credit department at 555-123-4567 for more information", title "Credit Terms?",name "Help"

Drivers: pdf

ARCHIVE

Syntax

archive “*libpath*”|{*expr*},”*doctype*”|{*expr*} ,”*docid*”|{*expr*} [,subid *subid*|{*expr*}] [,title *title*|{*expr*}] [,notes *notes*|{*expr*}] [,keywords *kws*|{*expr*}] [,categories|cats *cats*|{*expr*}] [,link|links *links*] [,entityid|entid *entity ID*|{*expr*}] [,args *args*|{*expr*}] [,dtm|date *yyyymmddhhmmss*|{*expr*}] [,subtitle *subtitle*|{*expr*}] [,subdtm|subdate *yyyymmddhhmmss*|{*expr*}]

Description

This command causes UnForm to add two versions of the current document to the library specified. The first document is a PDF version formatted as the current rule set specifies, the second is the input stream text from which the PDF document was generated. The PDF version has a default sub ID of “@UnForm”, but this can be overridden by specifying a *subid*. The sub ID of the text version is “@text”.

Note that as the formatted document is generated as a PDF, the rule set must be designed to successfully produce PDF output. In particular, any images or attachments need to be available in PDF format, or be designed to use automatic image conversion.

The elements of the archive command are evaluated as each page of the job is printed. If they change, then a sub-job is executed using the pages to that point as input. In this manner, a batch job with multiple documents can be archived as multiple documents rather than as a single large document. For example, if an invoice run is processed, and the *docid* is derived from the invoice number, then each new invoice number during the job will produce a new document in the archive.

The library is a path name on the UnForm server. If it doesn't exist, it will be created and a library pointer record will be created. If it isn't a full path, the library is created using the full path of the "arc" directory under the UnForm server, such as "/usr/lib/unform80/arc/*library*".

An archive document is identified in a library by the document type *doctype* and document ID *docid*. A document can also contain further information: title, notes, keywords, date and time, and categories. Further, each document can contain multiple versions identified by a *subid*, each of which contains a title and date and time. See the **Archiving and Document Management** chapter more information about each of the archive elements.

If any archive command elements are not supplied, the following defaults are used:

- The document type is set to the rule set name
- The document ID is set to a 10-digit sequential number
- The title is set to the value of the **title** command, if any, or is derived from text input
- Keywords are derived from unique words in the content
- The date and time is set to the current date and time
- Command line arguments, such as –arclib or –arcdoctype, supply remaining defaults

If the *subid* ends with an asterisk, such as “Formatted*”, then UnForm will not overwrite duplicate sub ID values. Instead, a 5-digit sequence will be added to ensure up to 99,999 versions of a document can be added.

If the keywords value begins with “*;”, the * is replaced with auto-derived keywords based on content, so you can have both auto and custom keywords using this structure. The keywords parameter can also be the word "all", or a number, indicating the maximum number of keywords to calculate (-1 means the same as 'all', and 0 means no keywords should be calculated).

Categories should be structured with vertical bars separating segments and semi-colons separating categories. For example: “CustPO|”+custname$+”|”+custpo$ + “;” + “Salesperson|”+slspid$. There can be any number of categories, and each category can contain up to ten segments.

Links provide outbound linking to other documents, within or without the archive system. This value is a semi-colon delimited list of links, each of which can be in one of the following formats:

- A full URL, optionally matching a URL used to load a document or image from a library, or a URL to an outside page or document. This structure, if it begins with http:// or ftp://, can be prefixed with a title in the format of *title=URL*. If the title is specified, that becomes the visible link in the browser.

- A simplified pipe-delimited structure of *library|doctype|docid*[|*subid*], which is displayed in the browser interface as a URL link to the document or image named by library, document type, document ID, and optionally image sub ID.

There can be any number of links in the list.

An entity ID can be set to tie this document to a particular code that can be used to filter access in the browser interface. A user login can be assigned to an entity ID, and that user can only view documents with a matching entity ID.

The args option can be used to specify UnForm command line arguments to pass to the sub-job used to generate the archive PDF file. For example, if you only want to archive copy 1 of a job, you could pass “-ce 1” (copies enabled 1).

If more than one archive command is present in a rule set, then archives are generated for all of them. For example, a second archive command might be added to produce a full job archive in addition to archives for individual documents. Note this differs from version 7.x, where only one archive command was honored.

Examples

```
archive "demo_accounting","ApAging"
```

This first example simply archives the A/P aging report to the demo_accounting library, under the document type “ApAging”. The document ID will be automatically generated as a 10-digit sequential number, and the entire job is archived as a single document. The title and keywords are derived automatically from the content.

```
archive "demo_sales","ArStatement",{arcid$},title {arctitle$},cats {arccats$}, args “-ce 1”
```

This example archives statements to the demo_sales library. The document ID, title, and categories are expressions derived from code block variables. The sub-job that generates the PDF document will have a “-ce 1” command line argument, which enables copy 1 only, so the archived copy will only be of the rule set’s first copy. The sample rule file arcdemo.rul contains the full Statement rule set where this example comes from.

Drivers: laser, pdf, ps

ATTACH

Syntax

attach "*filename*" | {*expr*}

Description

This will add the specified file to the output. The file will be added before any other text or data for a given copy is sent to the printer, so this can work as an overlay file, or it can be placed in the output instead of any text or other output, appearing like a stand-alone attachment.

If *expr* is used, then it should be a valid Business Basic expression that resolves to a string value, which will be interpreted as the file name as each copy prints.

When used as an attachment, assign a copy to the attachment, and use the **notext** keyword to suppress printing of text, like this:

```
if copy 1
 # the standard format
 # duplexing?  add duplex 1 in this copy
 text …
 box …
 etc…
end if

if copy 2
 # the attachment
 attach "/usr/UnForm/attach/attach1.pcl"
 notext
end if
```

When processing the file, UnForm will remove any printer initialization codes and page ejects from the file.

PCL Attachments
An easy way to create an attachment file is to use a Windows workstation and install a PCL5 type printer (not a PCL6 or PCL/XL driver, which will produce the wrong type of format). Set the port for the printer to FILE:. Then create the attachment using any word processor and print to that printer. Windows will ask for a file name, and when printing is complete, the resulting file is suitable for use as an attachment. If your document contains fonts that are not present in the printer you will be using, be sure to modify the print driver to print True Type Fonts as graphics, if possible.

PostScript Attachments
PostScript attachments are rendered simply as full page images, meaning the file can either be an EPS file or a JPG file (JPG files are only supported by color printers). UnForm simply prints the image, scaled to the printable region of the page.

PDF Attachments
UnForm attaches PDF documents by merging the objects on page one with those of the current page of output. Objects are placed in the exact same position and size as found in the attached document.

UnForm 8.0 supports PDF files up to version 1.4 (introduced with Acrobat 5). Some files that specify a later revision are still compatible, but new a file structure element was added at PDF 1.5 that is not supported by UnForm. Specifically, the unsupported feature is called an *Object Stream*. Former versions of UnForm did not support Linearized (also known as Optimized or Fast Web View) PDF files, nor files with incremental updates. Version 8.0 now supports these formats.

To create an attachment, use a PDF printer or other method to save a document in PDF format, choosing, if possible, to generate a file compatible with Acrobat 5 or below, or version 1.4 or below. There are many free and commercial tools available to produce PDF files, from Adobe and other vendors. Of note, Microsoft Office supports a "Save As" PDF feature with an Add-in that can be downloaded from Microsoft's web site.

PDF files are often available from third parties or government entities, and many of these are compatible with UnForm. Sometimes these files are designed with unusual page sizes or internal offsets that cause their elements to be in unexpected positions. UnForm is unable to re-position objects, so such files might need to be re-created using a PDF printer or other tool that will realign the objects using normal page dimensions.

Note that the object merging technique can cause issues when a landscape attachment file is designed to perform landscape formatting by rotating a portrait page, as UnForm executes landscape via a landscape page size rather than rotation. The result is a fundamental incompatibility between the two documents. To work around this, consider using UnForm commands to produce the document, or use the image command with a 'page *n*' option, supported when Ghostscript is available and configured.

Drivers: laser, pdf, ps

AUTHOR

Syntax

author "*authorstring*" | {*expression*}

Description

If this command is present, then PDF document creation adds an author *authorstring*, or the result of *expression*, to the document content. This value is available in the General Properties Display dialog in the Adobe Acrobat Reader.

Drivers: pdf only

BARCODE (PCL,PDF, PS)

Syntax

1. barcode *col*|{*numexpr*}, *row*|{*numexpr*},"*value*"|{*expr*},*symbology*,*height*,*spc-pixels* [,text] [,rotate *degrees*] [,start *char*] [,stop *char*] [,truncate] [,mode *mode*] [,cols *cols*] [,rows *rows*] [,usess] [,notrim]

2. barcode "*text*|~*regexpr*|!=*text*|!~*regexpr*[@*left*,*top*,*right*.*bottom*]", *col*|{*numexpr*}, *row*|{*numexpr*}, "", *symbology*, *height*, *spc-pixels*, getoffset *cols*, getcols *cols* [,eraseoffset *cols*] [,erasecols *cols*] [,text] [,rotate *degrees*] [,start *char*] [,stop *char*] [,truncate] [,mode *mode*] [,cols *cols*] [,rows *rows*] [,usess] [,notrim]

Description

col and *row* determine the upper left corner of the barcode. If used, *numexpr* is a Business Basic expression that generates a numeric value for the column or row.

value is a text string, up to 28 characters, to barcode (when using the Support Server, there is no limit on characters). Often this is symbology-dependent. If check digits are required, they are generated internally in UnForm. Within barcode families, if a unique symbology is associated with a specific length, then UnForm will internally select the correct symbology. For example, if a 9-digit zip code is specified with symbology 900 (5-digit post net), then symbology 905 will be used automatically.

Windows Support Server
Some options require the Windows Support Server for barcode image generation. The support server is bundled with Windows versions of UnForm, and available stand-alone (at no charge) to support Unix and Linux installations, assuming that the Unix/Linux system has network access to a Windows system running the support server.

If any of these options are used, or the usess option is provided, then UnForm will attempt to use the support server to generate the barcode.

The support server-only options are:

- text, which enables human readable text below the barcode
- rotate, which rotates the barcode image 90, 180, or 270 degrees
- start and stop characters, overriding the default Codabar characters of A and B, respectively
- truncate, which trims right edges off PDF417 barcodes
- mode value for Maxicode barcodes
    - 2 or 3 is for the transportation industry
    - 4 sets encoding to up to 93 characters or 138 digits
    - 5 encodes up to 77 characters with more error correction
    - 6 encodes programming messages for a scanner or reader

- cols and rows determine the dimensions of a PDF417 barcode (generally only cols is set, and rows is determined by the data to encode)

In addition, the support server is required to generate the following barcode symbologies:
- 950 Planet
- 960 OneCode (also called Intelligent Mail)
- 1000 PDF417
- 1100 Maxicode
- 1200 Data Matrix
- 1300 Aztec

expr is a Business Basic expression that generates the text to barcode.

symbology is one of the following numbers:

Code	**Description**
100	UPC VERSION A
105	UPC VERSION A + 2 DIGIT SUPPLEMENTAL ADD-ON
110	UPC VERSION A + 5 DIGIT SUPPLEMENTAL ADD-ON
125	UPC VERSION E
126	UPC VERSION E supporting number series 1, 6-digit input
130	UPC VERSION E + 2 DIGIT SUPPLEMENTAL ADD-ON
135	UPC VERSION E + 5 DIGIT SUPPLEMENTAL ADD-ON
150	UPC/EAN/IAN – 13
155	UPC/EAN/IAN – 8
200	INTERLEAVED 2 OF 5 – 2:1 CHECK DIGIT
205	INTERLEAVED 2 OF 5 – 2:1 NO CHECK DIGIT
220	INTERLEAVED 2 OF 5 – 3:1 CHECK DIGIT
225	INTERLEAVED 2 OF 5 – 3:1 NO CHECK DIGIT
300	STANDARD CODE 2 OF 5 – 2:1 CHECK DIGIT
305	STANDARD CODE 2 OF 5 – 2:1 NO CHECK DIGIT
320	STANDARD CODE 2 OF 5 – 3:1 CHECK DIGIT
325	STANDARD CODE 2 OF 5 – 3:1 NO CHECK DIGIT
400	CODE 39 (3 OF 9) – 2:1 NO CHECK DIGIT
405	CODE 39 (3 OF 9) – 2:1 CHECK DIGIT
410	CODE 39 (3 OF 9) – 2:1 NO CHECK DIGIT (FULL 128 ASCII)
415	CODE 39 (3 OF 9) – 2:1 CHECK DIGIT (FULL 128 ASCII)
440	CODE 39 (3 OF 9) – 3:1 NO CHECK DIGIT
445	CODE 39 (3 OF 9) – 3:1 CHECK DIGIT
450	CODE 39 (3 OF 9) – 3:1 NO CHECK DIGIT (FULL 128 ASCII)
455	CODE 39 (3 OF 9) – 3:1 CHECK DIGIT (FULL 128 ASCII)
500	CODE 93
600	CODE 128 – SERIES "A"

605	CODE 128 – SERIES "B"
610	CODE 128 – SERIES "C"
700	CODABAR – NO CHECK DIGIT
705	CODABAR – CHECK DIGIT
900	USPS Post net – 5 DIGIT
905	USPS Post net – 9 DIGIT
910	USPS Post net ABC – 11 DIGIT
The following symbologies require the Windows Support Server	
950	USPS Planet
960	USPS Intelligent Mail (Aka: OneCode, the 4-State Customer Barcode, 4CB and USPS4CB)
1000	PDF417 (2D)
1100	Maxicode (2D)
1200	Data Matrix (2D)
1300	Aztec (2D)

height is expressed in points or pixels. If it is an integer, such as 50 or 175, then it is treated as pixels at 300 dpi. If it is a floating-point number, like 18.7 or 12.0 (it contains a decimal point), then it is treated as points (1 point=1/72 inch). The maximum height is 3000 pixels.

spc-pixels is the number of pixels allocated to spacing between bars, from 1 to 50, the default being 2.

In syntax 2, triggered by a quoted value as the first argument, barcodes will be generated at all locations on a page where the *text* or the regular expression *regexpr* occurs. The value(s) to barcode will be based upon what text matches occur. Each match will determine the value to barcode based on the word found (up to the first space or the end of the line), and the placement of the barcode. The value to barcode can be adjusted by the getoffset *cols* (integer columns from the location of the match) and getcols *cols* (number of columns to use for the value). The location of the barcode can be adjusted by the *col* and *row* parameter, where 0,0 is the location where the match is found. The match text found can be erased from the report by setting eraseoffset *cols* and erasecols *cols*.

If the syntax "!=*text*" or "!~*regexpr*" is used, then the search is for positions NOT equal to the text or NOT matching the regular expression. When using the NOT syntax, only one search is performed per line in the search region.

The search for *text* or *regexpr* can be limited to a region on the page by adding a suffix in the format '*@left,top,right,bottom*'. To use a literal "@" character in *text* or *regexpr*, it is necessary to specify "\@".

Normally, UnForm will trim trailing spaces from the value being barcoded. If the notrim option is present, trailing spaces will not be trimmed. This can be critical if the barcode data contains spaces that must be retained. For example, a Code 128 barcode uses encoding that can produce space characters for valid data, so the notrim option should be used to prevent valid data from being truncated.

Version 5 Note: The positioning algorithm for PDF versions of the barcode was modified in Version 5 to match the positioning of laser barcodes. If your application depends on this older algorithm, then you can modify your ufparam.txt file (preferably by copying it to ufparam.txc and then modifying that file, to avoid losing your changes during an update) to add (or change) 'v4pdfbcd=1' in the [defaults] section.

Drivers: laser, pdf, ps

Examples:

barcode 10.5,22,{get(10,21,5)},900,12.0,2 will add a 12.0 point high, 5-digit post net barcode based on a zip code found at column 10, row 21.

barcode "bcd:@16,22,20,55",0,0,"",600,75,2, getoffset 4, getcols 10, erasecols 14 will search for data starting with "bcd:" in the region starting at column 16, row 22, through column 20, row 55, barcode the 10 characters following it, and erase the underlying text.

BARCODE (ZEBRA)

Syntax

barcode *col*|{*numexpr*}, *row*|{*numexpr*}, ("*value*" | {*expr*}), *symbology*, *height*, *spc-pixel*s, text [above|yes|no], rotate [90|180|270], ratio *rvalue*, checkdigit, start *startc*, stop *stopc*, ucc, mode *m*, security *s*, cols *c*, rows *r*, symbolno *val*, totsymbol *val*, chkhmn *val*, magfactor *val*, ecis *val*, errctrl *val*, menusymbol *val*, appendid *val*, model *val*, hqml *val*, nabk *val*, symboltype *val*, sepheight *val*, segwidth *val*, width39 *val*, ratio39 *val*, height39 *val*, heightpdf *val*, widthpdf *val*, quality *val*, escchar *val*

Description

col and *row* define the upper left corner of the barcode. If used, *numexpr* is a Business Basic expression that generates a numeric value for the column or row.

value is a literal value to barcode, *expr* is a Business Basic expression that generates the text to barcode.

symbology is one of:

Symbology	Name	Options Used
1	Code 11	rotate,checkdigit,height,text,above
2	Interleaved 2 of 5	rotate,height,text,above,checkdigit
3	Code 39	rotate,checkdigit,height,text,above
4	Code 49	rotate,height,text,mode
5	Planet	rotate,height,text,above
7	PDF417	rotate,height,security,cols,rows,truncate
8	EAN-8	rotate,height,text,above
9	UPC-E	rotate,height,text,above,checkdigit
A	Code 93	rotate,height,text,above,checkdigit
B	CODEABLOCK	rotate,height,security,cols,rows,mode
C	Code 128	rotate,height,text,above,checkdigit,mode
D	UPS Maxicode	mode,symbolno,totsymbol
E	EAN-13	rotate,height,text,above
F	Micro PDF417	rotate,height,mode
I	Industrial 2 of 5	rotate,height,text,above
J	Standard 2 of 5	rotate,height,text,above
K	ANSI Codabar	rotate,checkdigit,height,text,above,start,stop
L	LOGMARS	rotate,height,above
M	MSI	rotate,checkdigit,height,text,above,chkhmn
O	Aztec	rotate,magfactor,ecis,errctrl,menusymbol,symbolno,appendid
P	Plessey	rotate,checkdigit,height,text,above
Q	QR Code	rotate,model,magfactor,hqml,nabk
R	RSS (Reduced Space Symb)	rotate,symboltype,magfactor,sepheight,height,segwidth

S	UPC/EAN extensions	rotate,height,text,above
T	TLC39	rotate,width39,ratio39,height39,heightpdf,widthpdf
U	UPC-A	rotate,height,text,above,checkdigit
X	Data Matrix	rotate,height,quality,cols,rows,formatid,escchar
Z	Postnet	rotate,height,text,above

Many options are required only by certain symbologies. The options used are given in the table above. For details about use and required values for options, see the ZPL reference manual available from Zebra Technologies Corporation (http://zebra.com).

For Maxicode, you may specify a *mode* of 2 for UPS US addresses, 3 for UPS non-US addresses, or 4 for non-UPS coding (the default is 2). The data must consist of 2 segments:

Segment 1:

- Mode 2: 3-digit class of svc, 3-digit country code, 9-digit zip code
- Mode 3: 3-digit class of svc, 3-digit country code, 6-character zip code

Zebra requires this segment; the remaining segment format is specified by UPS.

Segment 2:

- Data content as required by UPS, starting with the "[)>"+$1E$ header.

For modes other than 2 or 3, segment 2 can contain variable content.

height is either an integer, interpreted as the number of pixels, or a decimal number, such as 20.0 or 40.6, interpreted as points (1/72 inch).

spc-pixels is the narrow bar width in pixels, from one to 10, defaulting to 2.

Following spc-pixels, the options can be in any order.

Rotate will rotate the barcode the given number of degrees.

Ratio will modify the wide bar to narrow bar ratio, from 2.0 to 3.0 in 0.1 increments. The default ratio is 2.0. Some symbologies have fixed ratios.

text or **text yes** will print the human readable value below the barcode. **text above** (or just **above**) will print this value above the barcode.

text no will not print the value, even if that is the default for the given symbology.

checkdigit will cause a checkdigit to be calculated and printed by the printer.

start *char* will set the start character, if used by the symbology.

stop *char* will set the stop character.

ucc will set the UCC Case Mode on code 128 barcodes.

mode *m* will set the mode code, which is symbology dependent. The UCC Case Mode may be set for code 128 with 'mode U'. The code 49 mode can be A for auto, or 0-5 as defined in the ZPL programmers' guide.

security *n* will set the security and/or error correction level for the PDF417 bar code. *n* can be a digit from 0 to 8.

cols *c*, **rows *r*** will set the cols and rows values for the PDF417 barcode. If not set, this barcode will assume a 1:2 row to column aspect ratio. *c* can range from 1 to 30, *r* from 3 to 90, and the product of *c* x *r* can't exceed 927.

For other options, see reference materials offered by Zebra Technologies Corporation (http://zebra.com).

Drivers: zebra only

BIN

Syntax

bin *bin-code*

Description

The **bin** keyword is used to specify the output bin for any copy. Larger, departmental laser printers often have two or more bins, allowing print job output to be separated. In UnForm, you can specify a bin for each copy, or for the whole job.

bin-code is printer-specific, with 1 generally being the top, face-down bin, and 2 being a side or rear face-up bin. Some models may offer additional bins; see your printer's documentation for additional bin codes.

The printer model's (-m command line option) PPD file (or generic pcl.ppd or ps.ppd files) can specify *OutputBin *bin-code* entries which are used if present.

Drivers: laser, ps

BOJ, BOP, EOJ, EOP

Syntax

{boj | bop | eoj | eop}"*text string*" | {*expr*}

Description

These keywords provide the ability to add escape codes to the beginning of the job (after the printer is initialized but before any data prints), before each page of each copy, after each page of each copy, and after the job ends, just before the printer is re-initialized.

The escape sequences can be entered as a quoted text string or an expression in braces.

When entering a text string, it is possible to include non-printable characters with angle bracket notation, such as "<27>&k10G", where "<27>" is used to include an escape character.

UnForm will normally provide all the control needed for a job. These keywords are included to handle unusual requirements, such as perhaps adding PJL coding to a job for special paper handling requirements.

An expression can take advantage of the getppd() function to load control sequences for PCL or PostScript out of the printer's PPD file (as specified by the –m command line argument or as the generic pcl.ppd or ps.ppd).

Prior releases supported an unquoted format for hex strings. UnForm no longer supports this syntax. If your rule file contains hex strings, convert the commands like these examples:

```
    boj 1b266c3247         change to      boj { ath("1b266c3247") }
    boj 1b 26 6c 32 47     change to      boj { ath(stp("1b 26 6c 32 47",3," ")) }
```

Examples:

This example shows adding PJL codes to a job, setting the title to "Title Of Job".

boj "<27>%-12345X@PJL<10>@PJL JOB NAME=<34>Title Of Job<34><10>@PJL ENTER LANGUAGE=PCL<10>"

Drivers: laser, ps only

BOLD, ITALIC, LIGHT, UNDERLINE

CBOLD, CITALIC, CLIGHT, CUNDERLINE

Syntax

1. bold|italic|light|underline *col*|{*numexpr*}, *row*|{*numexpr*}, *cols*|{*numexpr*}, *rows*|{*numexpr*}

2. bold|italic|light|underline "*text*|!=*text*|~*regexp*|!~*regexp*[*@left,top,right.bottom*]", *col*|{*numexpr*}, *row*|{*numexpr*}, *cols*|{*numexpr*}, *rows*|{*numexpr*}

If **cbold**, **citalic**, **clight**, or **cunderline** is used, then *columns* and *rows* are interpreted to be the opposite corner of the region, and columns and rows are calculated by UnForm.

Description

The region indicated by the *col*, *row*, *cols*, and *rows* parameters will have the indicated attribute (**bold**, **italic**, **light**, or **underline**) applied. All text in the input within that region, but not text generated by **text** keywords, will be affected. If used, *numexpr* is a Business Basic expression that generates a numeric value for the column, row, columns, or rows.

If syntax 2 is used, then the region is defined relative to any occurrence of the *text*, or of text that matches the regular expression *regexpr*. In these cases, there may be no affected regions, or several. c*olumn* and *row* are 0-based in these formats. The search for *text* or *regexpr* can be limited to a region on the page by adding a suffix in the format '*@left,top,right,bottom*'. To use a literal "@" character in *text* or *regexpr*, it is necessary to specify "\@".

If the syntax "!=*text*" or "!~*regexpr*" is used, then the search is for positions NOT equal to the text or NOT matching the regular expression. When using the NOT syntax, only one search is performed per line in the search region.

Note that the **font** command is a more powerful alternative to these commands, and it also offers support for fonts that support specific weights or styles other than these.

Examples:

bold 1,5,30,4 bolds a region from column 1, row 5, for 30 columns and 4 lines.

underline "TOTAL:",0,0,36,1 underlines a region beginning at a position where the text "TOTAL:" is found, extending for 36 columns. If "TOTAL:" isn't found, the keyword is ignored until the next page is analyzed.

Drivers: laser, pdf, ps. **underline** and **light** is supported on laser only. Not all pcl fonts support the **light** and **bold** options.

PostScript input not supported.

BOX, CBOX

Syntax

1. box *col*|{*numexpr*}, *row*|{*numexpr*}, *cols*|{*numexpr*}, *rows*|{*numexpr*} [,*thickness]* [,*shade*] [,*color*] [,rgb *rrggbb*] [,dbl|double [*gap*]] [,left *l*] [,right *r*] [,top *t*] [,bottom *b*] [,icols=*gridcols*] [,irows=*gridrows*] [,ccols=*gridcols*] [,crows=*gridrows*] [,lcolor=*color*] [,lcolor rgb=*rrggbb*] [,scolor=*color*] [,scolor rgb=*rrggbb*]

2. box "*text*|!=*text*|~*regexp*|!~*regexp*[*@left,top,right.bottom*]", *col*|{*numexpr*}, *row*|{*numexpr*}, *cols*|{*numexpr*}, *rows*|{*numexpr*} [,*thickness]* [,*shade*] [,*color*] [,rgb *rrggbb*] [,dbl|double [*gap*]] [,left *l*] [,right *r*] [,top *t*] [,bottom *b*] [,icols=*gridcols*] [,irows=*gridrows*] [,ccols=*gridcols*] [,crows=*gridrows*] [,lcolor=*color*] [,lcolor rgb=*rrggbb*] [,scolor=*color*] [,scolor rgb=*rrggbb*]

If **cbox** is used, then *columns* and *rows* are interpreted to be the opposite corner of the box, and columns and rows are calculated by UnForm.

Description

A box of the indicated dimensions will be drawn. All dimensions can be specified to 2 decimal places, in the range of -255 to +255.   Whole number *col* and *row* represent center points; lines are drawn to the center point of the character position identified in order to facilitate connections between lines. This differs from the **shade** keyword, which shades full character cells. It may be easier to use the **box** keyword's shade parameter than to calculate shade positions that are offset from similar box parameters. To draw lines rather than boxes, simply set the *cols* or *rows* to 1 or 0 (1 is a special rule maintained for historical reasons), or use the **line** command. If both *cols* and *rows* are 1, then a vertical line is drawn 1 character high. To draw a box that is 1 column wide or 1 row deep, use 1.01 or .99. If used, *numexpr* is a Business Basic expression that generates a numeric value for the column, row, columns, or rows.

If syntax 2 is used, then the box is drawn relative to any occurrence of the *text*, or of text that matches the regular expression *regexpr*. In these cases, there may be no boxes drawn, or several. *column* and *row* are 0-based in these formats and can be negative if required. The search for *text* or *regexpr* can be limited to a region on the page by adding a suffix in the format '*@left,top,right,bottom*'. To use a literal "@" character in *text* or *regexpr*, it is necessary to specify "\@".

If the syntax "!=*text*" or "!~*regexpr*" is used, then the search is for positions NOT equal to the text or NOT matching the regular expression. When using the NOT syntax, only one search is performed per line in the search region.

Line Thickness

The optional *thickness* parameter may be a number from 1 to 99, indicating the number of dots or pixels to use when drawing the box outline. The default thickness is 1 dot. UnForm always uses dots at 1/300 inch. If a shade parameter is desired, then the thickness parameter is required.

The left, right, top, and bottom options override the specified *thickness* for any given side of the box. Setting "left 0", for example, would erase the left side of the box, while "right 4" would set the right side to 4 pixels wide.

The double or dbl option indicates a double-lined box. Both the inner and outer lines will be drawn at the normal thickness, and the optional *gap* may be specified to set the pixels between each line. The default *gap* is 1 pixel. The *gap* must be a digit between 1 and 9.

Shading
The optional *shade* parameter may be used to specify a "percent gray" value from 1 to 100. Most laser printers can only print about 8 different shades of gray, so a value of 45, for example, may print the same pattern as 50. Note that if you specify a shade level of 0, this differs from not specifying any shade at all: a shade level of 0 will force a white interior, even if another box or shade command draws shading inside the bounds of the box. If an interior color is specified, shading is ignored. A shade value of -1 is equivalent to no shading at all.

Color
Color can be specified as "white", "cyan", "magenta", "yellow", "blue", "green", "red", or "black", or you can name an RGB value as a 6-character hex string with "rgb *rrggbb*", where *rr* is red (00-FF), *gg* is green (00-FF), and *bb* is blue (00-FF). To distinguish colors between the line and the shade region, use "lcolor" or "lcolor rgb" for lines, and "scolor" or "scolor rgb" for shade.

Grids
The *gridcols* and *gridrows* settings are used to draw grid lines and/or shade regions inside the box. *gridcols* specifies one or more vertical column settings in the structure of *column*[:*thickness*[:*shade*[:*color*|*rrggbb*]]]. Multiple columns can be delimited by any character other than digits, the decimal point (.), and the colon. Each column designates a vertical line to draw from the top to bottom edges of the outer box. If a thickness is specified, then the line is drawn using that thickness (0 would draw no line at all). The default thickness is 1. If shade is specified, then a shade region is drawn from the left edge or prior column. *gridrows* is identical in structure to *gridcols*, but specifies the horizontal rows rather than vertical columns. The "icols" and "irows" introducers indicate columns and rows relative to the upper-left corner of the outer box. The "ccols" and "crows" introducers indicate absolute columns and rows. In each case, any column or row specification outside the bounds of the box is ignored.

For partial shading, partial color shading, or multiple color shading, see the **shade** keyword. You can improve the look of shade regions on laser printers, especially at medium shade levels and 600 or higher dpi settings, by using the **gs** command.

Examples:

box 5.5,2.5,34,3,2,10 will draw a box 34 columns wide and 3 lines high, at column 5.5, line 2.5. The box border will be 2 dots wide (1/150 inch). It will be filled with 10% gray shading.

box 1,1,55,1 will draw a horizontal line, 55 columns wide, at column 1, line 1.

box "Customer Total",-1,-1,60,3 will draw a box around the text "Customer Total", beginning 1 column before and 1 row up, for 60 columns and 3 rows.

cbox 12,{start_row-.5},40,{end_row+.5} will draw a box with the top and bottom lines based on two numeric variables, which would have been previously calculated in a prepage or precopy code block. In using the **cbox** version, the second pair of numbers indicates the lower-right corner, rather than the number of columns and number of rows. The code block used to calculate these positions might look something like this code, which finds the first and last rows that contain any data in the row range of 22 through 55:

```
prepage{
start_row=0,end_row=0
for line=22 to 55
  if trim(text$[line])>"" then if start_row=0 then start_row=line
  if trim(text$[line])>"" then end_row=line
next line
}
```

cbox .5,22,80.5,66,3, ccols=10.5 30 55.5 67.5, crows=23.25:1:20 60 will draw a box from column 0.5, row 22 through column 80.5, row 66. The lines of this outer box will be 3 pixels wide. Inside this box will be vertical lines at columns 10.5, 30, 55.5, and 67.5. Also inside the box will be a 1 pixel high horizontal line at row 23.25, with 20% shading from row 22 to row 23.25, and another 1 pixel horizontal line at row 60.

Drivers: All (*gridcols* and *gridrows* options supported only in laser, ps, pdf), zebra only support 0% or 100% shading.

BOXR, CBOXR

Syntax

1. boxr *col*|{*numexpr*}, *row*|{*numexpr*}, *cols*|{*numexpr*}, *rows*|{*numexpr*} [*,thickness]* [,*shade*] [,*color*] [,rgb *rrggbb*] [,tl=*topleft*] [,tr=*topright*], [,bl=*bottomleft*], [,br=*bottomright*] [,icols=*gridcols*] [,irows=*gridrows*] [,ccols=*gridcols*] [,crows=*gridrows*] [,lcolor=*color*] [,lcolor rgb=*rrggbb*] [,scolor=*color*] [,scolor rgb=*rrggbb*]

2. boxr "*text*|!=*text*|~*regexp*|!~*regexp*[*@left,top,right.bottom*]", *col*|{*numexpr*}, *row*|{*numexpr*}, *cols*|{*numexpr*}, *rows*|{*numexpr*} [*,thickness]* [,*shade*] [,*color*] [,rgb *rrggbb*] [,tl=*topleft*] [,tr=*topright*], [,bl=*bottomleft*], [,br=*bottomright*] [,icols=*gridcols*] [,irows=*gridrows*] [,ccols=*gridcols*] [,crows=*gridrows*] [,lcolor=*color*] [,lcolor rgb=*rrggbb*] [,scolor=*color*] [,scolor rgb=*rrggbb*]

If **cboxr** is used, then *columns* and *rows* are interpreted to be the opposite corner of the box, and columns and rows are calculated by UnForm.

Description

A box with rounded corners of the indicated dimensions will be drawn. All dimensions can be specified to 2 decimal places, in the range of -255 to +255. Whole number *col* and *row* represent center points; lines are drawn to the center point of the character position identified in order to facilitate connections between lines. If used, *numexpr* is a Business Basic expression that generates a numeric value for the column, row, columns, or rows.

If syntax 2 is used, then the box is drawn relative to any occurrence of the *text*, or of text that matches the regular expression *regexpr*. In these cases, there may be no boxes drawn, or several. *column* and *row* are 0-based, in these formats, and can be negative if required. The search for *text* or *regexpr* can be limited to a region on the page by adding a suffix in the format '*@left,top,right,bottom*'. To use a literal "@" character in *text* or *regexpr*, it is necessary to specify "\@".

If the syntax "!=*text*" or "!~*regexpr*" is used, then the search is for positions NOT equal to the text or NOT matching the regular expression. When using the NOT syntax, only one search is performed per line in the search region.

Line Thickness
The optional *thickness* parameter may be a number from 1 to 99, indicating the number of dots or pixels to use when drawing the box outline. The default thickness is 1 pixel. UnForm always uses dots at 1/300 inch. If a shade parameter is desired, then the thickness parameter is required.

Corner Rounding
To specify the degree of rounding for different sides, specify values for tl, tr, bl, and br, as desired. The specification for each corner is *col*:*row*:*scale*, where *col* is the number of columns from the corner to

begin the rounding, *row* is the number of rows from the corner to begin rounding, and *scale* is the level of rounding, from –100 for fully convex, to 100 for fully concave, where 0 becomes a straight line from the column and row break points. If no rounding options are specified at all, then UnForm will apply default rounding to all four corners. If any rounding is specified, then any unspecified corners become square corners.

Shading
The optional *shade* parameter may be used to specify a "percent gray" value of from 1 to 100. Most laser printers can only print about 8 different shades of gray, so a value of 45, for example, may print the same pattern as 50. Note that if you specify a shade level of 0, this differs from not specifying any shade at all: a shade level of 0 will force a white interior, even if another box or shade command draws shading inside the bounds of the box.

Color
Color can be specified as "white", "cyan", "magenta", "yellow", "blue", "green", "red", or "black", or you can name an RGB value as a 6-character hex string with "rgb *rrggbb*", where *rr* is red (00-FF), *gg* is green (00-FF), and *bb* is blue (00-FF). To distinguish colors between the line and the shade region, use "lcolor" or "lcolor rgb" for lines, and "scolor" or "scolor rgb" for shade.

Grids
The *gridcols* and *gridrows* settings are used to draw grid lines and/or shade regions inside the box. *gridcols* specifies one or more vertical column settings in the structure of *column*[:*thickness*[:*shade*[:*color*|*rrggbb*]]]. Multiple columns can be delimited by any character other than digits, the decimal point (.), and the colon. Each column designates a vertical line to draw from the top to bottom edges of the outer box. If a thickness is specified, then the line is drawn using that thickness (0 would draw no line at all). The default thickness is 1. If shade is specified, then a shade region is draw from the left edge or prior column. *gridrows* is identical in structure to *gridcols*, but specifies the horizontal rows rather than vertical columns. The "icols" and "irows" introducers indicate columns and rows relative to the upper left corner of the outer box. The "ccols" and "crows" introducers indicate absolute columns and rows. In each case, any column or row specification outside the bounds of the box is ignored.

For partial shading, partial color shading, or multiple colors shading, see the **shade** keyword. You can improve the look of shade regions on laser printers, especially at medium shade levels and 600 or higher dpi settings, by using the **gs** command.

Zebra Printers
Zebra output supports rounded corner boxes somewhat differently than laser/pdf output. All corners have the same scale of rounding, so the first corner option (tl, rt, bl, br) is used for all corners. The scale value must be a number from 1 to 8, indicating the scale of rounding the printer performs. For example, tl=::6 would apply a rounding scale of 6. The col and row parameters of the corner specification are ignored.

Examples:

boxr 10,9.5,70,4.25,2,5,lcolor=blue will draw a box with default rounding on all corners, with a 2 pixel edge and 5% shading. The edge line will be drawn in blue if the output device supports color.

cboxr 0.5,60,80.5,66,1,0,bl=3:1.5:75,br=3:1.5:75 will draw a box with corners 0.5,60 and 80.5,66, with a 1 pixel border, no shading, and just the bottom left and right corners rounded. The rounding will start 3 columns and 1 row from the corners, and be rounded outward.

Drivers: laser, pdf, ps (laser cannot have –nohpgl specified), zebra (see notes)

CIRCLE

Syntax

1. circle *col*|{*numexpr*}, r*ow*|{*numexpr*},*radius*|{*numexpr*} [,*thickness*] [,*shade*] [,color|lcolor *colorname*] [,scolor *colorname*] [,color|lcolor rgb *rrggbb*] [,scolor rgb *rrggbb*]

2. circle "*text*|!=*text*|~*regexp*|!~*regexp*[*@left,top,right.bottom*]", *col*|{*numexpr*}, r*ow*|{*numexpr*},*radius*|{*numexpr*} [,*thickness*] [,*shade*] [,color|lcolor *colorname*] [,scolor *colorname*] [,color|lcolor rgb *rrggbb*] [,scolor rgb *rrggbb*]

Description

A circle with the center at the column and row specified, with the radius specified, will be drawn. All dimensions can be specified to 2 decimal places, in the range of -255 to +255. If used, *numexpr* is a Business Basic expression that generates a numeric value for the column, row, and radius. The radius is specified as a number of columns. For a fixed measure radius, use an expression with the inchtocols() or cmtocols() function.

If syntax 2 is used, then the circle is drawn relative to any occurrence of the *text*, or of text that matches the regular expression *regexpr*. In these cases, there may be no circles drawn, or several. *column* and *row* are 0-based, in these formats, and can be negative if required. The search for *text* or *regexpr* can be limited to a region on the page by adding a suffix in the format '*@left,top,right,bottom*'. To use a literal "@" character in *text* or *regexpr*, it is necessary to specify "\@".

If the syntax "!=*text*" or "!~*regexpr*" is used, then the search is for positions NOT equal to the text or NOT matching the regular expression. When using the NOT syntax, only one search is performed per line in the search region.

Line Thickness
The optional *thickness* parameter may be a number from 1 to 99, indicating the number of dots or pixels to use when drawing the box outline. The default thickness is 1 pixel. UnForm always uses dots at 1/300 inch. If a shade parameter is desired, then the thickness parameter is required.

Shading
The optional *shade* parameter may be used to specify a "percent gray" value of from 1 to 100. Most laser printers can only print about 8 different shades of gray, so a value of 45, for example, may print the same pattern as 50.

Color
Color can be specified as "white", "cyan", "magenta", "yellow", "blue", "green", "red", or "black", or you can name an RGB value as a 6-character hex string with "rgb *rrggbb*", where *rr* is red (00-FF), *gg* is green (00-FF), and *bb* is blue (00-FF). To distinguish colors between the line and the shade region, use "lcolor" or "lcolor rgb" for lines, and "scolor" or "scolor rgb" for shade.

Examples

The following will draw a circle centered in an 80 by 66 form, with a 2.5 inch radius, a blue 2-pixel wide border, and a 5 percent interior shade.

```
circle 40.5,33,{inchtocols(2.5)},2,5,lcolor blue
```

Drivers: laser, pdf, ps (laser cannot have –nohpgl specified)

COLS

Syntax

cols *n*

Description

This keyword specifies the number of columns to use for the form or report. The base font is scaled to accommodate this many columns. If present, this value will override any calculation based on the **cpi** keyword.

The number of columns *n* can be any value up to 255.

Examples:

cols 80 will set the print pitch to accommodate 80 columns per page.

Drivers: all

COMPRESS, NOCOMPRESS

Syntax

compress
nocompress

Description

If zlib support is available on the UnForm server (most operating systems support it), then UnForm will use the “deflate” compression method by default. This produces very compact PDF files. If you do not wish to produce such compressed files (for example, you want to see the PDF commands contained in the file), then you can use the **nocompress** option (or the –nocompress command line option) to turn off this default compression mode.

If no zlib support is available, the **compress** command can be used to use the RLE compression algorithm. This is most effective when repeated characters like spaces are present in the output, such as wide reports with empty space between columns. Compression requires extra processing and will therefore affect performance.

Compression can also be turned on with the –compress command line option.

You can determine if zlib support is enabled by viewing the version information produced by the uf80c –v command line.

Drivers: pdf only

CONST, GLOBAL, LOCAL

Syntax

const|global|local *ID*=*value*

Description

The **const** keyword provides the capability to use a named value as a parameter to other keywords. If, for example, you want to place a series of text values at a certain column position, but may need to adjust the position in the future, and then set a constant *ID* to the column position *value*, then use the *ID* in the column position of all the text values.

```
const COLPOS=22.25
text COLPOS,30,"Text line 1"
text COLPOS,31,"Text line 2"
text COLPOS,32,"Text line 3"
```

A given constant ID can be reused, and references to it in subsequent rule set lines will reflect the new value. Also, a constant defined before the first rule set in the rule file will apply to any rule sets in the file, unless the same ID is reused in any particular rule set. The **global** command may be used in place of const for one of these pre-rule set constants. Likewise, the **local** command can be used in place of const inside a rule set.

Note that case does make a difference. "COLPOS" and "colpos" are different constants. Take care not to use constant names that may inadvertently cause unintended replacements. For example, it may be tempting to use a constant named "font", but this would conflict with any font command. There would be no conflict, however, between a constant named FONT and a lower-case font command.

Constant names are limited to 255 characters, and constant values are limited to 65,536 characters. If you use a quoted value, the outer quotes are removed before the value is substituted into the rule file commands. You can therefore include quotes inside a quoted constant. Unquoted values are trimmed of leading and trailing spaces.

Long constant values can be built by including the constant name in multiple const commands, like this:

```
    Const VAL=”Initial Value”
    Const VAL=”VAL plus this appended data”
    Const VAL=”VAL and still more appended data”
```

Drivers: all

COPIES, PCOPIES

Syntax

copies *copies*
pcopies *copies*

Description

These keywords are used to generate multiple copies of the form. The number of copies is specified by the number *copies*. If the **copies** form is used, then the entire print job is duplicated the number of times indicated. If the **pcopies** form is used, then each page is duplicated as it is printed, so the pages come out collated together for each page.

The two versions of this keyword are mutually exclusive; the last one that is found in the rule set is the one used. Note also the **-c** and **-pc** command line options can be used, though these keywords take precedence, if specified.

Individual copies can be managed to any degree necessary via "if copy *n*" rule set logic, and also full programming logic with the "precopy {}" and "postcopy {}" logic entry points. Use this to modify the output device for specific copies, or to modify the content of specific copies.

To add attachments that are separate pages from the standard form pages, assign a copy to the attachment, and add a **notext** keyword for that copy.

```
pcopies 2

if copy 2
notext
attach "/usr/UnForm/attachments/attach1.pcl"
end if
```

Examples:

copies 2 will print the entire report twice.

pcopies 3 will print each page three times.

Drivers: all, pdf driver treats copies as pcopies

COVER

Syntax

cover "*ruleset*"|{*expr*} [, "*rulefile*"|{*expr*} [,"*args*"|{*expr*}]]

Description

Processes the named rule set (optionally in a different rule file) as a subjob, using the first page of text of the current job as the input stream. The resulting one page of output is used as an initial page of the main job. Both pcl and ps output will generate cover pages for each output file when the job is broken into multiple output designations. If arguments are specified, they are passed to the subjob, in addition to the -r/-f options named by the ruleset and rulefile options.

In addition to the cover command, the -cover command line argument, as well as the coverset$, coverfile$, and coverargs$ code block variables, can be used to generate cover pages. Also, setting nocover=1 in a code block will disable cover page generation. This can be used to turn off the effect of a -cover command line option.

Example

This example will generate a cover page from the corpcover rule set in covers.rul, passing it a name and number parameter. The corpcover rule set could retrieve the name and number in a prejob code block, using prm("name") and prm("number").

```
cover "corpcover","covers.rul",{"-prm "+quo+"name="+name$+";number="+faxnum$+quo}
```

CPI

Syntax

cpi *characters-per-inch*

Description

The **cpi** keyword indicates what pitch UnForm should use when printing the text of a form or report. From this, along with the paper dimensions, UnForm can determine the columns per page and ensure that the proper pitch is selected. As UnForm uses **cpi** to calculate a **cols** value, **cpi** values are rounded to allow even character spaces. It is advisable to use **cols** rather than **cpi**.

See also **lpi**, **cols**, **rows**.

Examples:

cpi 16.66 will set the character spacing to a common "compressed" character pitch.

Drivers: laser, pdf, ps, zebra

CROSSHAIR

Syntax

crosshair

Description

If this command is present in a rule set, then UnForm will generate a crosshair grid over the page, making rule file development easier. Crosshair mode can also be turned on from a code block with the crosshair$ variable.

Drivers: laser, pdf, ps

DELIVER

Syntax

deliver "*send to*"|{*expr*},"*docid*"|{*expr*} [,combine yes|true|1 |{expr}] [,args "*list*"|{*expr*} [,*tag* "*value*"|{*expr*}, ...]

Description

The deliver command generates subjobs whenever the document ID and output format changes, producing either fax or email files as needed based on the list of "send to" values. The output format is determined by the deliver.ini file, and if the recipient is an email or fax destination. Multiple destinations can be provided in a comma-separated list and each will get their own copy of the document. If the combine option is on (yes, true, or 1 turn it on), then all documents for a given destination are combined into a single transaction. Note that not all fax systems offer support for multiple documents in a single transaction (i.e. msfax). The document ID is used as the basis for the file name to be sent, which can be useful when emailing to provide meaningful attachment file names.

The args option specifies command line options passed to the subjob. The deliver.ini configuration can also add more options. Other tag names are used to substitute values in the delivery gateway's configuration lines in deliver.ini. More details are provided in that file, and in the Deliver Configuration chapter.

Any number of tags can be specified, using user-defined tag names. When the delivery is executed, the deliver.ini configuration is scanned and occurrences of %*tagname* are substituted with the provided value.

Multiple delivery commands can be used. This may be desired if different arguments, such as cover pages or -prm parameters, are desired to distinguish email from fax jobs for formatting. If this is the case, the document ID should also vary, since only a unique ID and output type cause a new subjob to be executed.

When the subjob is executing, both uf.subjob and uf.deljob are true (1).

CSV formatted logs are maintained in the ./deliver directory (or other configured directory named in logdir= in deliver.ini. The logs are named yyyymmdd.csv and record date/time, to, file, success, response and error messages, and optionally the tags.

Note that code blocks can also use the deliver() function, managing the file to be delivered with jobstore/jobexec functions or other techniques.

Example

```
deliver {faxnum$},{"Invoice "+invno$},name {contact$},subject {"Invoice "+invno$},
note "Your invoice is attached.\n\nThank you for your business."
```

DETECT

Syntax

detect *column(s)*,*row(s)*,"[^[!]]*text"*
detect *column*(s),*row(s)*,"[^[!]]~*regexpr*"

Description

This option is used to identify a form from the data read by UnForm. If the **-r** option is used on the UnForm command line, then **detect** keywords are ignored. Otherwise, each rule set's detects are analyzed until a match is found. If more than one **detect** keyword is specified for a rule set, then the job must match all of them. Detection occurs only at the start of the job, using the first page of data read from the input stream.

If *column* and *row* are 0, then the whole page is scanned for the occurrence of the text. If *column* is 0 and *row* is greater than 0, then the whole line is scanned. If column is greater than 0 and row is 0, all rows are scanned.

column and *row* can contain ranges in the format *from-through*, such as '20-25' for the columns (or rows) 20 through 25.

The format of the quoted third parameter determines how the detection scan is handled. If plain text is specified, then a literal match for *text* is performed. If the text begins with the prefix character ~, then a regular expression search for *regexpr* is performed.

If the text begins with ^, then a case insensitive match is performed.

Following the optional ^ character, but before the ~ character, may be a ! character, indicating a scan for NON-matches.

The following prefix sequences are valid: ^, ^~, !, !~, ^!, ^!~, meaning, respectively: case insensitive text, case insensitive regular expression, text not found, regular expression not found, case insensitive text not found, case insensitive regular expression not found.

DTC Rule Sets

When the UnForm Desktop Client processes detect statements, it honors certain options. For details, see the Desktop Client rule set section.

Examples:

detect 0,2,"INVOICE" would search for INVOICE anywhere on line 2.

detect 10-12,4,"~../../.." would match a date format at column 10, 11, or 12, on row 4.

detect 65-66,6-8,"!~../../.." would match a date format NOT occurring at column 65 or 66, on rows 6 through 8.

detect 0,2-3,"^invoice" would match INVOICE, Invoice, invoice, etc. anywhere on lines 2 or 3.

Drivers: all

DOWN

Syntax

down *n* [,*gap*]

Description

This instructs UnForm to allocate virtual pages down the physical page, evenly spaced within the top and bottom margins. Use this feature for multi-up printing of standard reports, or for laser labels.

UnForm will automatically scale text (to as small as 4 point), boxes, and shading. It will not scale images, barcodes, or attachments. Also see the **across** command.

Down can be used inside an 'if copy' block, but is only compatible with non-collated copies. As a result, copy-specific **down** is only available in the laser driver, and only in conjunction with the **copies** command, not **pcopies**.

If the optional *gap* value is specified, it indicates the number of vertical pixels between each virtual page. If it is not specified, the default is to use 1 *row* (as opposed to pixels).

See the 132x4 rule set in advanced.rul for an example of using the across and down commands.

Drivers: laser, pdf, ps

PostScript input not supported

DPI

Syntax

dpi 300 | 600 | 1200

Description

The **dpi** keyword instructs PCL printers to print at the specified dots per inch. The default dpi value is 300; however, many printers are capable of printing at 600 or 1200 dpi (or possibly even higher values). This takes more printer memory, but results in crisper characters and lines.

Drivers: laser only

DSN_SAMPLE

This command is used exclusively by the UnForm Designer tool, to store the name of a sample text file to apply to previews generated in the design environment.

DTCBUTTON

Syntax

DTCButton "*title*" [,style clipboard|nbclipboard|text|nbtext|button|title|nbtitle] [,args "*job arguments*"] [,width *chars*] [,match "*regex*"] [,library "*name*"] [,doctype "*value*"] [,parsevalue ["*ruleset*"]]

Description

The DTCButton command is interpreted by the UnForm Desktop Client for presentation in an application integration window. For each DTCButton command, an input field is constructed, along with a submission button, and presented on the form associated with the rule set. The DTCPanel command can be used to categorize the buttons in panels. Input fields are presented in the same order as presented in the rule set. Note that expressions are not supported.

Style
This optional argument defines the style of the button provided to the user. The default style is "clipboard".

Clipboard	Provides a text box and a small submit button, and monitors the clipboard for changes to place text in the text box.
Nbclipboard	Provides a text box, but no submit button. It monitors the clipboard for changes to place in the text box.
Text	Provides a text box and a small submit button. This is intended for simple user entry.
Nbtext	Provides a text box, but no submit button.
Button	Provides a submit button with the title as the button caption.
Title	Provides a text box and a small submit button, and monitors the window title for changes to place in the text box.
Nbtitle	Provides a text box, but no submit button. It monitors the window title for changes to place in the text box.

Args
Options passed to the to the UnForm job running the rule set. There are automatic rule file and rule set arguments (-f and –r, respectively) to ensure the rule set with the button configuration is the one executed when a submission takes place.

Width
This defines the width of the button, in nominal characters. Without a width, the title width is used. Use this to ensure consistent widths when multiple buttons are presented.

Match

The match option is honored by the clipboard and nbclipboard styles. The clipboard value must match the specified regular expression in order to be pasted into the text field.

Library, Doctype
These two options in tandem are honored by the clipboard and nbclipboard styles. The clipboard value must be a valid document ID within the library and doctype specified in order to be pasted into the text field.

ParseValue
If this option is present, the value from the clipboard or window title is sent to the server for parsing. The server will run the current rule set or the optionally specified one, and use the value returned from the server in cgiresponse$ in place of the clipboard or title value. The rule set receives cgi.button$, cgi.panel$, and cgi.parsevalue$ when the request is sent.

DTCHELPFILE

Syntax

DTCHelpFile "*filename*"

Description

When the UnForm Desktop Client presents an application integration form (defined using DTCButton and DTCPanel commands), there is a toolbar help button available. If a DTCHelpFile command specifies a file, the help button is enabled, and when pressed, the HTML file specified is loaded in a browser window on the DTC user's workstation. The help file must be in HTML format, and must be available to the UnForm server using normal HTTP interface locations, in the ./web/en-us or ./web/*language* path, in the home UnForm directory.

DTCPANEL

Syntax

DTCPanel "*panelname*"

Description

The DTCPanel command can specify a panel name for DTCButtons that follow. The buttons will be presented within a tab panel, using *panelname* as the title. This command is interpreted by the UnForm Desktop Client.

DUMP

See the **image** command.

DUPLEX

Syntax

duplex *mode* [, *left-offset*] [, *top-offset*]

Description

Duplex printing, if supported by your printer, causes printing on both sides of the paper.

mode can be 1 for long-edge binding, or 2 for short-edge binding. A *mode* of 0 will print in simplex (single-sided) mode.

left-offset and *top-offset* are optional values in decipoints (1/720th inch) that indicate how far to shift the page printing from the left and top edges, respectively. Note that margins may need to be adjusted (with the **margin** keyword) if offsets are used.

Note that any duplex command will cause a page eject on a laser printer, so timing of the duplex command is important. For example, if you use pcopies 2, and the second reserved for a back side attachment, the duplex command should be in the 'if copy 1' block. This forces copy 1 to be on the front side and copy 2 to follow on the back side. This concept is shown in the example below.

The printer model's (-m command line option) PPD file (or generic pcl.ppd or ps.ppd files) can specify *Duplex *mode* entries which are used if present.

Note that when working with multiple copies to produce pages or unique formats in duplex mode (i.e. terms and conditions on the back page of primary forms), only the **pcopies** command will work, as it is critical that the copies be printed in sequence rather than at a job-level.

Examples:

```
pcopies 2
if copy 1
 duplex 1
 # complete form for front of page
end if
if copy 2
  # attachment for back of page
  notext
  attach "terms.pcl"
end if
```

Drivers: laser, ps (the left offset and top offset options are ignored in PostScript, use margin instead)

EMAIL

Syntax

email { *to* | {*toexpr*} }, { *from* | {*fromexpr*} }, { *subject* | {*subjectexpr*} }, { *msgtxt* | {*msgtxtexpr*} } [,cc "*cc*"|{*ccexpr*}] [,bcc "*bcc*"|{*bccexpr*}], [,attach "*attach*"|{*attachexpr*}] [,otherhead|oh "*otherhead*"|{*otherheadexpr*}] [,login "*login*"|{*loginexpr*}] [,password|pswd "*password*"|{*passwordexpr*}] [,logfile *filename*]

Description

The PDF document being created will be emailed as an attachment upon completion, using the information supplied. The name of the attached file is supplied with the "-o" argument on the UnForm command line, or can be overridden by setting the variable output$ in a prejob code block.

Each of the first 4 values is positional, and each can be a literal value or an expression enclosed in curly braces. The *to* value is the only required value, and must be a fully qualified email address, or a comma-separated list of email addresses. The *from* value, if supplied, must also be a fully qualified email address. If it is not supplied, then a default address will be used from the mailcall.ini file.

Note that the expressions are resolved as of the last copy of the last page of the job. If you need to use data from an initial page, use a prejob code block to assign variables, and then use those variables in the expressions.

In order to use this command, the mailcall.ini file must be edited to configure a mail server (server=*value*) line. See the Email Integration chapter for more detail about configuration, and also for information about using direct calls to the MailCall program bundled with UnForm. Direct calls enable more control over email processing.

The *msgtxt* value can contain line-feed characters to break lines. These characters can be added in expressions as CHR(10) functions or as $0A$ hex literals, or with the literal backslash-n (\n) character sequence. Note that if the message text starts with a structure "<*value*>", then it is assumed to be an HTML message, and the appropriate header tag is set to send the message as HTML.

Optional arguments can follow the message text value in any order, prefixed by the appropriate option name:

cc	Followed by a literal that is, or an expression in curly braces that resolves to, a list of email addresses separated by commas. These addresses become the CC, or carbon copy, list for the email.
bcc	Followed by a literal that is, or an expression in curly braces that resolves to, a list of email addresses separated by commas. These addresses become the BCC, or blind carbon copy, list for the email. Blind carbon copy addresses are stripped from the email header before the message is sent.

attach	Followed by a literal that is, or an expression in curly braces that resolves to, a list of additional attachment files, separated by commas. Note that the PDF job itself is always emailed as an attachment, so only use this option for adding additional attachments to the message.
otherhead or oh	Followed by a literal that is, or an expression in curly braces that resolves to, one or more line-feed or "\n" delimited custom email headers.
login	Followed by a literal that is, or an expression in curly braces that resolves to, a login name. Some mail servers are configured to require a login and password for authentication. This value and the password value are then required.
password or pswd	Followed by a literal that is, or an expression in curly braces that resolves to, a login password. Some mail servers are configured to require a login and password for authentication. This value and the login value are then required.
logfile	Followed by a file name to which SMTP logging will be written.

Example

```
prejob{
email_to$=trim(get(1,1,50))
invoice_no$=get(60,5,6)
}

email {email_to$}, "sales@acme.com", {"Invoice number "+invoice_no$}, "Please pay the attached
invoice promptly.\n\nBest regards,\n\nAcme Distributing", cc "accounting@acme.com"
```

Drivers: pdf only

ERASE, CERASE

Syntax

1. erase *col*|{*numexpr*}, *row*|{*numexpr*}, *cols*|{*numexpr*}, *rows*|{*numexpr*}

2. erase "*text*|!=*text*|~*regexp*|!~*regexp*[*@left,top,right.bottom*]", *col*|{*numexpr*}, *row*|{*numexpr*}, *cols*|{*numexpr*}, *rows*|{*numexpr*}

If **cerase** is used, then *columns* and *rows* are interpreted to be the opposite corner of the region, and columns and rows are calculated by UnForm.

Description

The text from the input, in the region indicated by the *column*, *row*, *columns*, and *rows* parameters, is erased. This keyword may be used to easily clear unwanted text from the output. The text is erased after text expressions and prepage and precopy code blocks are executed, so the information to be erased is available to those routines. If used, *numexpr* is a Business Basic expression that generates a numeric value for the column, row, columns, or rows.

If syntax 2 is used, then the region is defined relative to any occurrence of the *text*, or of text that matches the regular expression *regexpr*. In these cases, there may be no erased regions, or several. *column* and *row* are 0-based in these formats. The search for *text* or *regexpr* can be limited to a region on the page by adding a suffix in the format '*@left,top,right,bottom*'. To use a literal "@" character in *text* or *regexpr,* it is necessary to specify "\@".

If the syntax "!=*text*" or "!~*regexpr*" is used, then the search is for positions NOT equal to the text or NOT matching the regular expression. When using the NOT syntax, only one search is performed per line in the search region.

Also see the erase option of the **hline** and **vline** keywords.

When erase is used with PostScript input, it is converted internally to a **shade** command with a shade percent of 0, resulting in erasure of the region from the overlay. Rule set output commands, such as text or box, are layered on top of the erased region.

Examples:

erase 1,5,30,4 erases text from a region from column 1, row 5, for 30 columns and 4 lines.

erase "John Smith",0,0,10,1 erases all occurrences of "John Smith" from the page.

Drivers: all

FIXEDFONT

Syntax

fixedfont *fontcode*

The **fixedfont** keyword overrides the default fixedfont setting found in the [default] section of the ufparam.txt file. If there is no fixedfont value in that file, then the *fontcode* 4099 (Courier) is used.

The *fontcode* specified is used for the text sent to UnForm by the application. It must be a non-proportional, scalable font, except in the circumstance where a non-scalable font provides the exact pitch required by UnForm to lay out the columns within the margins.

Drivers: laser only

FONT, CFONT

Syntax

1. font *col*|{*numexpr*}, *row*|{*numexpr*}, *cols*|{*numexpr*}, *rows*|{*numexpr*} [,*fontname*] [,font *fontcode*] [,symset *symset*] [,*size*] [,bold] [,italic] [,underline] [,light] [,shade *percent*] [,fixed | proportional] [,*color*] [,rgb *rrggbb*] [,*justification*] [,upper|lower|proper] [,fit] [,weight *w*|*weightname*] [,style *style*|*stylename*] [,column *n*]

2. font "*text*|!=*text*|~*regexp*|!~*regexp*[@*left,top,right.bottom*]", *col*|{*numexpr*}, *row*|{*numexpr*}, *cols*|{*numexpr*}, *rows*|{*numexpr*} [,*fontname*] [,font *fontcode*] [,symset *symset*] [,*size*] [,bold] [,italic] [,underline] [,light] [,shade *percent*] [,fixed | proportional] [,*color*] [,rgb *rrggbb*] [,*justification*] [,upper|lower|proper] [,fit]] [,weight *w*|*weightname*] [,style *style*|*stylename*] [,column *n*]

If **cfont** is used, then *columns* and *rows* are interpreted to be the opposite corner of the region, and columns and rows are calculated by UnForm.

Description

The **font** keyword applies font control to all input stream text in the defined region of column, row, columns, and rows. The other parameters are all optional. If used, *numexpr* is a Business Basic expression that generates a numeric value for the column, row, columns, or rows.

If syntax 2 is used, then font attributes are applied relative to the occurrence of *text* or the regular expression *regexpr*. In these cases, there may be no attribute regions, or several. *column* and *row* are 0-based in these formats, and can be negative if required. The search for *text* or *regexpr* can be limited to a region on the page by adding a suffix in the format '*@left,top,right,bottom*'. To use a literal "@" character in *text* or *regexpr*, it is necessary to specify "\@".

If the syntax "!=*text*" or "!~*regexpr*" is used, then the search is for positions NOT equal to the text or NOT matching the regular expression. When using the NOT syntax, only one search is performed per line in the search region.

Font Names and Numbers
fontname can be Courier (the default), CGtimes, or Univers. These fonts are standard on virtually all PCL5 compatible printers. Alternately, font *fontcode* can specify a specific fontcode supported by your printer. For example, if your printer supports True Type Arial, specify "font 16602". Bitmap fonts (as opposed to scalable fonts) should not be used. *fontname* and *fontcode* can also be specified from the "ufparam.txt" file. UnForm uses HP/GL by default for laser output, and justification is supported on all native printer fonts. However, if the –nohpgl command line option is used, then only certain, known fonts (found in fonts.txt in the UnForm directory) can be properly justified, if the center, decimal, or right *justification* option is used. When producing PDF output, only native PDF fonts are supported. All others are mapped to one of these fonts: Courier, Helvetica, or Times-Roman.

Symbol Sets
symset can be any symbol set supported by your printer. The default symbol set is "9J", using a Windows ANSI character set. *symset* can also be a name from the "ufparam.txt" file. The pdf driver only supports the Windows ANSI symbol set.

Point and Pitch Sizes
size is a numerical value that specifies the point size of a proportionally spaced font or the pitch size of a fixed font. Values range from about 4 to 999.75. The default is based on the rows per page. Note that for proportional fonts, the larger the number, the larger the size printed. Fixed fonts are the opposite.

Attribute Styles
The words "bold", "italic", "underline", and "light" will apply the indicated attribute(s) to the text.

Shaded Text
percent indicates the percent gray to print the text, from 0 (white) to 100 (black). The default is black.

Fixed and Proportional Text
Any font code below 4100 is presumed to be fixed (mono-spaced), and codes 4100 and up are presumed to be proportional. To override this assumption, specify one of the words "fixed" or "proportional".

Color
Color can be specified as "white", "cyan", "magenta", "yellow", "blue", "green", "red", or "black", or you can name an RGB value as a 6-character hex string with "rgb *rrggbb*", where *rr* is red (00-FF), *gg* is green (00-FF), and *bb* is blue (00-FF).

Justification
justification can be one of the following words: "left", "center", "right", or "decimal". UnForm will remove leading and trailing spaces from the text and justify it within the column specification. Decimal justification will use a "." character unless a "decimal=*character*" line is placed in the ufparam.txt file under the [defaults] section.

Text Case Conversion
The mutually exclusive "upper", "lower", and "proper" options will convert the text in the fonted region to all UPPER, lower, or Proper case. Proper case capitalizes the initial letter of each word or word segment preceded by a non-letter or non-digit character.

Fit to Width
If the "fit" option is used, then each line in the font region is scaled down, if necessary, to fit within the defined number of columns for the region. This differs from the text command's fit option, in that each line is treated distinctly, rather than the entire set of lines being calculated as a unit.

Weight and Style
Some laser printer fonts must be specified with given weight or style in order to be selected by the printer. For example, the font Clarendon Condensed is only available if the condensed style is specified, by adding "style 4" or "condensed" to the font command. Style and weight options and codes can be

found in the ufparam.txt file. Note that fonts are expressly designed for certain weights and styles, and simply specifying an unsupported value does not produce the desired result. In fact, it may result in selection of a different font entirely. Check your printer's documentation or control panel prints for supported fonts.

Note that if you use identical font commands for two adjacent or overlapping regions, UnForm will combine the regions. For proportionally spaced fonts, the result will be misaligned columns. To avoid this, you can add non-operational options, like "black" or "shade 100" to alternating commands, so UnForm will not treat them as identical. Alternatively, use the column option (8.0.04), specifying a unique column between 0 and 222 to prevent region combining.

Examples:

font 10,20,29,50,cgtimes,12,center will change the text in the region starting at column 10, row 20, for 29 columns and 50 rows, to 12-point cgtimes. The text will be centered within the 29 column width.

cfont 1,20,132,52,courier,16.67 will change the font of the region specified to 16.67 pitch courier. Since courier is a mono-spaced font, the number 16.67 is interpreted as a pitch (characters per inch) rather than a point size.

cfont {pos("Description"=text$[22]},23,{pos("Units"=text$[22])-1},60,univers,10 will calculate the starting and ending column based upon where "Description" and "Units" occur in line 22, and change the font for that column range, for rows 23 through 60.

Drivers: all, but note the following:

PDF: maps pcl font names and numbers to Courier, Helvetica, or Times-Roman. Symbol set 9J is the default and the only symbol set supported.

Ps: maps pcl font names and numbers to a setting defined in the [psmap] section of ufparam.txt. Matching Type1 font files can be installed in the psfont directory.

zebra: symbol sets are not supported. *size* is limited to scalability of the font in the printer's firmware, typically integer multiples of the base font size in dots. Color is not supported, nor is justification. Shading can be either 100% (black) or 0% (white). Font names are not mapped. Specify fonts instead as font codes, which must be internal font identifiers, such as a-f, 0-9. See the ZPL documentation for font codes.

The fit option is only supported in laser, ps, and pdf drivers.

PostScript input not supported.

GS

Syntax

gs [yes | on | no | off]

Description

The **gs** command can be used to control graphical shading. The command by itself or followed by the words "yes" or "on" will turn on graphical shading. Any other parameter value will turn graphical shading off, resulting in the highly efficient, though not as finely rendered, internal laser shade commands. The –gs command line option can be used to specify graphical shading by default.

If dpi is set to 600 or above (and the printer supports 600 dpi printing), graphical shading is even more finely rendered. Note that some faxing products that convert pcl code into low-density bitmaps provide more readable output without graphical shading. You can selectively turn graphical shading on or off within "if copy" blocks.

Using the **gs** command will add approximately 2000 bytes of additional overhead to a job.

Example:

```
gs on

if copy 2
  gs off
  output {"|vfx –n " + faxnumber$ +" –F pcl"}
end if
```

Drivers: laser only

HLINE

Syntax

hline "*text*" [,erase] [,extend] [,*thickness*]

Description

Any horizontal occurrence of the *text* indicated, of at least the length indicated, will be replaced with a horizontal line. The *text* must be composed of a single character repeated any number of times. There can be multiple **hline** keywords in a rule set, if needed. For example, if both dashes (-) and equal signs (=) are used for lines in a form, both can be specified in separate **hline** keywords.

This keyword is useful if the application already produces boxes and lines with standard characters. Also see the **vline** keyword.

As with all box drawing, UnForm will consider line endpoints to be at the center position of a character, which may impact how lines intersect. Lines are drawn 1 pixel (1/300 inch) thick.

If the "erase" option is used, then no line is drawn. Instead, the horizontal text values are simply removed from the output.

If the "extend" option is specified, the lines are extended ½ character left and right. The *thickness* parameter specifies a pixel width to draw.

The search for *text* can be limited to a region on the page by adding a suffix in the format '*@left,top,right,bottom*'. To use a literal "@" character in *text*, it is necessary to specify "\@".

Example:

hline "---" will search the report for 3 or more horizontal dashes. All such dashes found will be replaced with a horizontal line.

Drivers: all

PostScript input not supported.

HSHIFT

See the **shift** command.

IF COPY … END IF

Syntax

if copy *n,n,…*
…
end if

Description

The **if copy** command will cause any following commands, up to an **end if** command, to apply only to the copy or copies specified. The feature is used to manipulate the content of various copies. For example, you may wish to add a text message on a specific copy, or suppress a region of text with a white shade. When combined with **attach** and **notext** keywords, attachments can be added without the printing of text.

end if indicates that conditional processing of the rule set is done, and keywords apply to all copies again. The **end if** keyword may also be entered as **endif** or **fi**.

Examples:

if copy 2 will process keywords following this line, until an **endif** keyword is found, and apply keywords only to copy 2.

if copy 3,4,6 will apply keywords to the 3 copies identified.

Drivers: all

IF DRIVER … END IF

Syntax

if driver *name*
…
end if

Description

The command **if driver** will cause any commands to apply only when the rule set is evaluated under the driver *name*. The driver is specified with the command line option "-p", and defaults to "laser". Common drivers are laser, pdf, ps, and zebra. If Ghostscript drivers are configured, then other driver names are available based on the [drivers] configuration in the server's uf80d.ini file.

The command 'if driver zebra' applies to any Zebra driver specification (Zebra drivers include suffix-based information).

end if indicates that conditional processing of the rule set is done, and keywords apply to all copies again. The **end if** keyword may also be entered as **endif** or **fi**.

Example:

This example will use the image "pdflogo.pdf" when "-p pdf" is used on the command line.

```
if driver pdf
  image 1.5,2,15,6,"pdflogo.pdf"
end if
```

Drivers: all

IF *EXPRESSION* … END IF

Syntax

if *expression*
…
end if

Description

The if *expression* block test evaluates the *expression* as Business Basic syntax to determine if the enclosed UnForm rule set commands should be included in the current job. Rule set parsing occurs after the command line is parsed, but before the job is executed. The expression can therefore use uf.*xxx* values and access -prm values via the gbl() or prm() functions.

end if indicates that conditional processing of the rule set is done, and normal parsing continues. The **end if** keyword may also be entered as **endif** or **fi**.

Examples

```
if uf.pdftitle$=""
  title "Default Title"
end if
```

```
if prm("email")>""
  # command line contained -prm email=xxx
  email {prm("email")},...
end if
```

Drivers: laser, pdf, ps, zebra

IMAGE

Syntax

image *col*|{*numexpr*}, *row*|{*numexpr*} [, *cols*|{*numexpr*}, *rows*|{*numexpr*}], "*file*" | {*expr*} [,color], [,nocache] [,option *code*] [,shade *percent*] [,gamma *gamma*] [,rotate *rotate*][,page *pagenum*|{*expr*}

Description

The **image** command is used to print an image file specified by *file* or the *expr* which resolves to a file name to each page when the output position is the *column* and *row* indicated. This option is typically used to add graphic logos to forms. The column and row can be specified with decimal fractions to 1/100 character.

The optional *cols* and *rows* parameters are used in most circumstances. Specifically, they are not used for native PCL images (.pcl or .rtl files), or for raw Zebra images (.zpl). In those cases, the columns and rows options are ignored. For Postscript and PDF images, and for images that are scaled and converted for use in PCL output, columns and rows are required. If not supplied in those circumstances, then each defaults to the cols and rows that measure one inch. It is generally advisable to include these parameters to ensure that all versions of output will produce the desired size whenever possible.

If used, *numexpr* is a Business Basic expression that generates a numeric value for the column, row, columns, or rows.

If *expr* is used, then it should be a valid Business Basic expression that resolves to a string value, which will be interpreted as the file name as each copy prints.

Images typically require some degree of processing or parsing during a print job. To avoid extra overhead for images that are used repeatedly, the final images are cached for reuse in later jobs. However, in some cases it may be preferable to avoid caching images. For example, signature images are often used only once, and caching them would waste space and cache management processing time. To turn off caching of an image, use the nocache option. Note that cached images that remain unused for a period of time (defined by imageage=*days* in uf80d.ini) are removed from the cache file (images.dat) automatically.

The color option indicates that a color image should be produced. The –ci and –color command line options set the color option on by default for all images. This option is ignored when printing native format images, such as .pcl or .rtl images, but it may determine what file name substitutions might occur in some output formats.

The shade option can be used to apply shading to the image to reduce it's intensity and allow other data to show through the image. In PDF output, transparency must be enabled for this to work.

The gamma value can be used to modify the colors when an image is being converted to native format. Images that contain RGB color information often display differently on different devices and as a result can appear darker or lighter than expected when printed. A gamma value greater than 1 lightens an image, while a value less than 1 darkens it. This value is passed to the conversion program (Image Magick or the Windows Support Server). It is ignored when using image files that are already in native format for the output.

Rotation can occur when an image is being converted to native format. This value is passed to the conversion program (Image Magick or the Windows Support Server). It is ignored when using image files that are already in native format for the output.

The page option is supported for PDF images and PDF output. See the discussion below.

Image handling varies considerably based on the output format, and the type of image provided to the image command. Image files can be considered "native", meaning they can be inserted by UnForm directly into the output with little or no processing, or they can be other image formats that require conversion first. Conversion is automatic, assuming either Image Magick or the Windows Support Server are configured and available for UnForm's use.

The publisher also maintains an image conversion tool at http://unform.com. This tool allows users to upload images in non-native format and produce native images for use within UnForm jobs.

PCL Output

UnForm considers an image with a .pcl, .prn, or .rtl extension to be a native PCL image. These images are parsed to remove PCL instructions that could cause a page eject, and positioning code is inserted at runtime to place the image in the correct location on the page. Otherwise, the image is passed through to the output unchanged.

If an image is not in native PCL format, such as a .jpg or .tif file, then it must be converted at runtime to PCL format. During this conversion process, the image is also scaled to the size required. The conversion and scaling process is accomplished with Image Magick, if configured in the uf80d.ini file, or via the Windows Support Server, if configured in uf80d.ini or with a sshost() code block command. Internally, the image is first converted to a bitmap in color or black and white format, and then resolved into a PCL image internally.

Notes on Native PCL Images
Some PCL images contain width and height information. However, since passing through a width and height would apply a default crop size to any additional image without width and height information, UnForm strips out this width and height coding. Unfortunately, color PCL images must contain width and height information to prevent the printer from displaying a black band from the right edge of the image to the right margin. Therefore, when printing to a color printer, UnForm must pass the image width and height coding to the printer. To trigger this parsing behavior, you can do one of two things: add a "color" option to the image command, or add a –gw, -ci, or –color option to the UnForm command line.

One side effect of passing this coding through is that you can't also use images that do not have size information. Generally, this means you can't mix color and black and white PCL image files in the same job.

If the *row* is 0 or 255, then UnForm will apply no positioning to the output. In this case, the positioning desired should be present in the file. UnForm will scan the file, looking for image information and possibly position data. Just that information will be sent to the output device. If the row is greater than 0 and less than 255, then UnForm will ignore any positioning that might be contained in the image file, and instead place the upper left corner of the image where specified. This feature only works with pcl images that include positioning data.

PDF Output
If UnForm is producing PDF output, and the image file name ends in .pcl, .prn, or .rtl, then the file name is modified to have a .pdf extension automatically. This allows a single fixed file name to accommodate both laser and PDF output without special logic.

Because PDF files can contain various image formats, including full-color (24-bit) jpeg files, and supports both color and black and white image data, UnForm goes through several logical steps to determine how best to insert the image.

- If the image file extension is .pcl, .prn, or .rtl, UnForm instead looks for a file of the same name, but with a .pdf extension. If the file is not found, a warning error is issued and no image is produced. Otherwise, native .pdf handling is performed.

- If the image is in .pdf format, and the 'page *n*' option is not used, UnForm parses the file for the first image on the first page, and inserts that image object in the output. Note that UnForm's parser supports PDF files revision 1.4 and below.

  If the 'page *n*' option is used, then UnForm will use Ghostscript 8.10 or higher to produce a scaled image of the specified page. This page image will be in either color or black and white format, depending on whether color or black and white image output is being produced. If Ghostscript is not configured in the uf80d.ini file or via the Windows Support Server, this option results in a warning message and no page image is produced.

- If the image command or command line indicates color output, and the image file extension is jpg or jpeg, Unform checks its size and colors setting. If it is a 24-bit jpeg file and its size is no more than 50% larger than the specified size (based on cols and rows), then the file is inserted directly into the output.

- If no conversion facilities are available (no Image Magick configuration and no Windows Support Server), and the image is a 24-bit jpeg file, it is inserted into the output.

- UnForm attempts to convert and scale the image file into either a black and white pdf image, or a 24-bit color jpeg image, depending on the whether the output is color or not.

Due to internal buffer management, PDF file names can't exceed 75 characters.

PostScript Output
PostScript image output must be in either eps or jpeg format. Black and white laser printers do not support jpeg images. Color laser printers support both eps files and both gray scale (8-bit) and color (24-bit) jpeg images.

UnForm performs these logical steps to determine how to insert the image:

- If the image file extension is .pcl, .prn, or .rtl, then the file name is modified to have either a .eps extension (for black and white output) or a .jpg extension (for color output). If the file exists, the new file name is processed.

- If the file extension is jpeg and the output is color, the file is inserted in the output.

- If the file extension is eps, the file is inserted in the output.

- Other image formats are converted to jpeg or a black and white PostScript raster image, using either Image Magick or the Windows Support Server. If these facilities are not available, a warning error is issued and the image is not inserted.

Zebra Output
UnForm scans the first block of the file to determine if it is a native ZPL image file. ZPL images begin with the characters "~DG" followed by some comma-delimited data. If so, the image is inserted in the output.

If the file is not a native ZPL image, UnForm attempts to convert it to a bitmap using Image Magick or the Windows Support Server, and then to a ZPL image. During this process, the image is scaled to the size specified by the cols and rows parameters. If the conversion fails, then a warning error is issued and no image is inserted.

Zebra does not support color or shaded images.

Examples:

image .5,1.25,"/usr/UnForm/logo.pcl" will place the raster image contained in the named file at column .5, row 1.25.

**image {icol},{irow},{icols},{irows},{logo$}** will place an image file specified in the variable logo$ at the position specified by the variables icol and irow. If used in a pdf driver or when automated conversion and scaling is invoked, the variables icols and irows would specify the image size (more specifically, its bounding box) in columns and rows. All the variables would have to be created in a code block, such as prejob{} or prepage{}.

Drivers: all.

IMAGES

Syntax

images "*filelist*"|{*expr*} [,across *n*] [,down *n*] [,res|resolution *n*] [,color] [,tray value|{*expr*}]

Description

Appends image files from the filelist, which can be a literal or an expression. The list may contain any number of semi-colon delimited file names. Each is converted to a native image in sequence and added to pages following the current page, optionally scaled and tiled based on the across and down options.

The images are produced at 300 dpi unless otherwise specified. The images will by default be produced in black and white unless pcl color images are indicated with a uf80c –color (or –ci) command line option, or if the color option is specified in the images command.

Note that images, particularly color images and high resolution images, can become quite large, resulting in larger print jobs or pdf files than are typical.

Images must be scalable, so therefore images need to be in a format that can be converted via configured image conversion and scaling programs or the UnForm **Windows Support Server**. Further, of course, one of these image conversion options must be enabled. Common formats include jpg, tiff, bmp, and png. Specifically, pcl images cannot be used as they are not scalable. In addition, if Ghostscript is available on the server or the **Windows Support Server**, then pdf files can be included. This is particularly useful when used with UnForm archive libraries, as pdf images can be extracted from the libraries and appended to a job. See the Statement example in samples/arcdemo.rul for an example of using the images command along with functions to extract archive images as PDF files.

The images command can involve a great deal of image processing. Performance benefits can be achieved with PDF files if Ghostscript 8.10 or higher is available, either on the UnForm server (indicated by the pdffitpage=1 entry in uf80d.ini) or via the Windows Support Server.

The tray option is provided to draw paper from the tray specified for the pages produced by the image attachments. The tray value should match the syntax found in the **tray** command, or can be an expression that produces an equivalent text value.

An images command may be placed inside an 'if copy' block or be run with an exec() command in a code block in order to append images only on selected copies or pages.

Examples

images “termsconditions1.jpg;termsconditions2.jpg”

images {all_invoices$},across 2, down 2, resolution 150, tray 5

Drivers: laser, pdf, postscript

ITALIC

See the **bold** keyword.

JAVASCRIPT

Syntax

```
javascript "text"|{expr}
```

Description
`
This command adds document-level javascript to the pdf document. This code is executed as the document opens, so can be used to invoke actions when the document is opened or to define functions for use in annotation actions specified with a javascript: url in the **annotate** command.

Examples

```
javascript “function showMessage(msg) { alert msg; }”

prejob{
crlf$=$0d0a$
js$=”function showMessage(msg)”+crlf$+”alert msg;”+crlf$+”}”
}
javascript { js$}
```

Drivers: pdf only

KEYWORDS

Syntax

keywords "*keywordstring*" | {*expression*}

Description

If this command is present, then PDF document creation adds a keyword *keywordstring*, or the result of *expression*, to the document content. This value is available in the general properties display dialog in the Adobe Acrobat Reader.

Drivers: pdf only

LANDSCAPE, RLANDSCAPE

Syntax

landscape or rlandscape

Description

This keyword will ensure that UnForm produces output in landscape (horizontal) orientation. The default orientation is portrait (vertical), unless UnForm encounters a PCL control code setting landscape mode (hex 1B266C314F) on the first page. Use of this keyword will force landscape mode regardless of PCL control codes found in the input.

The **rlandscape** command will turn on reverse landscape mode.

Note that landscape is supported inside 'if copy' blocks, allowing different copies to be in different orientations.

Also see the **portrait** keyword.

Drivers: laser, ps, pdf (rlandscape is laser only)

LCOPIES

Syntax

lcopies *n* | {*expr*}

Description

Setting this value will cause UnForm to add a thermal printer copies command (^PQ in Zebra) to the print output, specifying the printer generate *n* or numeric expression *expr* duplicate labels. This is different than using an UnForm **copies** or **pcopies** command, as it instructs the printer to generate duplicate labels at the printer, rather than allowing for distinct formatting for different copies. The benefit of using this command is that the copies are produced by the hardware and not the print stream output, so overhead is reduced and performance is higher.

Alternatively, in a prepage or prejob code block, the variables lcopies$ or zcopies$ can be set to a numeric string (i.e. lcopies$=str(10)).

Drivers: zebra

LDARKNESS

Syntax

ldarkness *n* | {*expr*}

Description

Setting this value will cause UnForm to send a thermal printer darkness command (~SD in Zebra) to the printer. The darkness parameter can be a number *n* or a numeric expression *expr*. The value of *n* should be an integer between 0 and 30, based upon current ZPL documentation.

Alternatively, in a prepage or prejob code block, the variables ldarkness$ or zdarkness$ can be set to a numeric string (i.e. ldarkness$=str(5)).

Drivers: zebra

LIGHT

See the **bold** keyword.

LINE

Syntax

1. line *col1*|{*numexpr*}, r*ow1*|{*numexpr*}, *col2*|{*numexpr*}, r*ow2*|{*numexpr*} [,*thickness*] [,color|lcolor *colorname*] [,rgb *rrggbb*]

2. line "*text*|!=*text*|~*regexp*|!~*regexp*[@*left,top,right.bottom*]", *col1*|{*numexpr*}, r*ow1*|{*numexpr*}, *col2*|{*numexpr*}, r*ow2*|{*numexpr*} [,*thickness*] [,color|lcolor *colorname*] [,rgb *rrggbb*]

Description

A line is drawn between the first column and row and the second column and row. All dimensions can be specified to 2 decimal places, in the range of -255 to +255. If used, *numexpr* is a Business Basic expression that generates a numeric value for the columns and rows.

If syntax 2 is used, then the line is drawn relative to any occurrence of the *text*, or of text that matches the regular expression *regexpr*. In these cases, there may be no lines drawn, or several. c*ol1* and *row1* are 0-based, in these formats, and can be negative if required, and *col2* and *row2* are the number or columns and rows to draw from the offset position. The search for *text* or *regexpr* can be limited to a region on the page by adding a suffix in the format '*@left,top,right,bottom*'. To use a literal "@" character in *text* or *regexpr*, it is necessary to specify "\@".

If the syntax "!=*text*" or "!~*regexpr*" is used, then the search is for positions NOT equal to the text or NOT matching the regular expression. When using the NOT syntax, only one search is performed per line in the search region.

The origin point of a line is the center of a cell. In other words, position 1,1 is the center of the first character cell, 0.5,0.5 is the upper left corner of the cell, and 1,1 is the lower right corner of the cell. This is consistent with the box command.

This positioning is different from version 7.0, requiring that column positions be shifted 0.5 columns left, and row positions be shifted 0.5 rows up.

Line Thickness
The optional *thickness* parameter may be a number from 1 to 99, indicating the number of dots or pixels to use when drawing the box outline. The default thickness is 1 pixel. UnForm always uses dots at 1/300 inch. If a shade parameter is desired, then the thickness parameter is required.

Color
Color can be specified as "white", "cyan", "magenta", "yellow", "blue", "green", "red", or "black", or you can name an RGB value as a 6-character hex string with "rgb *rrggbb*", where *rr* is red (00-FF), *gg* is green (00-FF), and *bb* is blue (00-FF).

Examples

The following will draw a light blue line from the upper left corner to the lower right corner of a page:

```
line 0.5, 0.5, 80.5, 66.5, 1, rgb 008000
```

The following will underline the word “TOTAL”, in conjunction with a font command:

```
font “TOTAL:”,0,0,5,1,cgtimes,12
line “TOTAL:”, 0, .75, {textwidth(“TOTAL:”, “cgtimes”,12,0)},0
```

Drivers: laser, pdf, ps (laser cannot have –nohpgl enabled)

LOAD

Syntax

load "*filename*"

Description

The load command is similar, but not identical, to the merge command. Its purpose is to insert rule set text from a disk file. The entire contents of the file is loaded into the rule set in place of the load command.

This differs from merge in that the merge command loads a specific rule set by its name (identified in the file as [*name*], whereas the load command loads the entire file.

LOCKCOLS

Description

When the label dimensions and cols setting are established, UnForm scans mono-spaced internal fonts for the closest match that will not exceed the cols specified, then recalculates the cols to agree with the font selected. This allows print stream text and all other enhancements to scale together. However, it also causes labels to shrink in printable area width, sometimes very noticeably, resulting in graphical commands not being placed where expected. This option was added to prevent this recalculates from occurring, at the expense of losing the print stream scale matching. With this option, graphical commands will print where expected on the label, but may not align with print stream output.

Drivers: zebra only

LPI

Syntax

lpi *line-height*

Description

The **lpi** keyword indicates the vertical line height UnForm should use when printing the text of a form or report. From this, along with the paper dimensions, UnForm can determine the rows per page and ensure that the proper vertical placement is selected for each line. To save time and effort, use the **rows** keyword and UnForm will calculate the lpi.

See also **cpi**, **cols**, **rows**.

Examples:

lpi 8 sets 8 lines per inch.

lpi 6.6 uses a common laser printer value based on 66 lines in a 10 inch printable page length on letter paper.

Drivers: all

LSPEED

Syntax

lspeed *n* | {*expr*}

Description

Setting this value will cause UnForm to send a thermal printer speed command (^PR in Zebra) to the printer. The value of *n* or result of *expr* should be an integer between 2 and 6, or between 8 and 12, based upon printer documentation.

Alternatively, in a prepage or prejob code block, the variables lspeed$ or zspeed$ can be set to a numeric string (i.e. lspeed$=str(2)).

Drivers: zebra

MACRO

Syntax

macro *n*

Description

This keyword will cause UnForm to invoke macro number *n* in the LaserJet printer. This macro must be defined and downloaded to the printer as a permanent macro. This keyword could be used to call a macro for a company letterhead, for example. For more information, see the Working With Macros chapter.

Drivers: laser only

MACROS

Syntax

macros on|off

Description

This keyword causes UnForm to invoke (or not invoke) macros for fixed raster elements (**box**, **shade**, **text**, **image**, and **attach**). Macro usage can significantly reduce the data transfer requirements to the printer, most noticeably on a serial or parallel connection with many pages of similar output. The printer must have enough memory to store and execute the macros.

The default macros setting is "off"; the "-macros" command line option establishes the default macros setting to "on". This keyword overrides either default for this rule set.

Macros are numbered from 0 to 32767. UnForm will start macro definitions at 32000 unless the "[defaults]" section, "macrono" field is set to a different value in the ufparam.txc file. If a site uses macros and finds a conflict with this number, then the value should be changed to allow an available contiguous range for UnForm.

Drivers: laser only

MARGIN

Syntax

margin[s] *left*, *right*, *top*, *bottom*

Description

The **margin** keyword is used to increase the margins used by UnForm when calculating row and column positions. Normally, UnForm will use a 0.25 inch margin on all 4 sides, based on the paper size in use. If you need to increase any margin, you can specify the dot offsets desired. Note that the values for *left*, *right*, *top*, and *bottom* are entered in dots, which default to 300 dpi, but can be modified by the **dpi** keyword.

For example, **margin 75,75,0,150** (at 300 dpi) would set left and right margins to 0.5 inches, the top margin would remain at 0.25 inches, and the bottom margin would be 0.75 inches.

Drivers: laser, pdf, ps

MERGE

Syntax

merge "*ruleset*" [, "*rulefile*"]

Description

This command will insert the contents of the *ruleset* into the currently parsed rule set. If the *rulefile* parameter isn't supplied, the current rule file is used. Otherwise, *rulefile* is opened in the UnForm directory or by full path, if specified, and is scanned for *ruleset*. This command can be used to incorporate common elements into many rule set formats. For example, a name and address heading could be placed into a rule set called "address_header", and various forms could use the command **merge "address_header"** to include the commands it contains.

Note that if no *rulefile* is specified, then the rule file specified for the job is used for the merge, even if the merge is nested within another merge that specifies another rule file.

Unlike other UnForm commands, **merge** works within code blocks, such as precopy or prepage, as well as outside of code blocks.

Drivers: laser, pdf, ps, zebra

MICR

Syntax

micr *col*|{*numexpr*}, *row*|{*numexpr*}, *"account"*|{*expr*}, *"check"*|{*expr*}

Description

Prints MICR font at the *col* and *row* specified, for laser check printing. If used, *numexpr* is a Business Basic expression that generates a numeric value for the column and row. The account number must be in the format **:123456789:xxx"**, where the colons surround the 9-digit bank number, and the balance of the account number is terminated with, or contains, a quote. Quotes can be identified in a text literal with <34>. A space after the bank number and terminating colon is optional. When the MICR code is generated, colons or A become a "transit" symbol, B becomes an "amount" symbol, quote or C become an "on us" symbol, and a hyphen or D becomes a dash . Account numbers can contain these symbols, spaces, and digits. The check number can be up to 12 digits long. This keyword supports 8 inch checks only, not the smaller 6 inch variety, which requires a different format for the MICR.

If no "on us" symbol is present in the account number (i.e. no <34> or C character), then one is appended automatically.

The fixed bank number is typically hard-coded, but can be an expression if enclosed in braces {}. The check number will generally be an expression, which can use get() to retrieve the number from the application print, or can be a variable defined in a prepage{} block.

Note that with proper soft font configuration, you can use the text command to print MICR encoded data in any format, such as that required by a deposit slip. The same MICR soft fonts included for use with this command can be used as text soft fonts.

Example:

micr 6,42.25,":123456789:9999-1234<34>",{trim(get(65,5,6))} would print a MICR encoded line with the indicated bank and account number, and a check number derived from the input stream data printed at column 65, row 5, for 6 characters.

Drivers: laser, ps only

MOVE, CMOVE

Syntax

1. move *col*|{*numexpr*}, *row*|{*numexpr*}, *cols*|{*numexpr*}, *rows*|{*numexpr*}, *newcol*|{*numexpr*}, *newrow*|{*numexpr*} [,retain]

2. move "*text*|!=*text*|~*regexp*|!~*regexp*[@*left,top,right.bottom*]", *col*|{*numexpr*}, *row*|{*numexpr*}, *cols*|{*numexpr*}, *rows*|{*numexpr*}, *movecols*|{*numexpr*}, *moverows*|{*numexpr*} [,retain]

Description

cmove causes *cols* and *rows* to be interpreted as the opposite corner of the region to be moved.

The **move** keyword moves a block of text to a new location on the page. Syntax 1 moves the region indicated by *col, row, cols,* and *rows* so the new upper left point is at *newcol, newrow*. Syntax 2 searches for occurrences of *text* or the regular expression *regexpr,* respectively, and uses each location found as a point from which *col* and *row* are measured (0-based movement). The rectangular region specified is then moved *movecols* left or right, and *moverows* up or down. The search for *text* or *regexpr* can be limited to a region on the page by adding a suffix in the format '*@left,top,right,bottom*'. To use a literal "@" character in *text* or *regexpr*, it is necessary to specify "\@".

If used, *numexpr* is a Business Basic expression that generates a numeric value for the column, row, columns, or rows, and also the "new" column and row (syntax 1) and the "move" columns and rows (syntax 2).

If the syntax "!=*text*" or "!~*regexpr*" is used, then the search is for positions NOT equal to the text or NOT matching the regular expression. When using the NOT syntax, only one search is performed per line in the search region.

The optional "retain" parameter will cause UnForm to leave the text in its original location, in effect copying the text rather than moving it.

Move commands simply shift text around in an internal array, so it is possible for moves to cascade to other moves. Moves that specify positions (syntax 1) are performed in the order found in the rule set, then moves that are relative to text (syntax 2) are performed in the order found in the rule set.

Note that **move** commands occur *after* any **shift** or **vshift** commands. If you would like to move data based on positions before the **shift** and **vshift** commands, consider using a **text** command with an expression using the cut() or mcut() functions.

Examples:

move 5,10,40,4,20,20 moves text at column 5, row 10, 40 columns wide and 4 rows high, to the region 20,20,40,4.

move "Date",0,0,4,1,-4,0 moves all occurrences of the word Date left by 4 columns.

Drivers: laser, pdf, ps

PostScript input not supported.

NOTEXT, NOOVERLAY

Syntax

notext or nooverlay

Description

This keyword specifies that no report text or graphical print stream data should be printed. Typically, this would be placed inside an "if copy *n*" block in order to add an attachment and prevent overwriting of the form text.

Example:

```
if copy 2
        attach "/usr/UnForm/attachments/attach1.pcl"
        notext
end if
```

Drivers: all

OUTLINE

Syntax

outline [*level*]

Description

The **outline** keyword turns on the production of PDF outlines (also called bookmarks) and the automatic display of the outline when the document is displayed in an Adobe Acrobat Reader. The content of the outline is set page by page, by setting the variable "outline$" in a precopy or prepage code block. Multi-level outlines can be specified by delimiting the levels with vertical bar (|) characters in the outline$ string.

If *level* is supplied, it must be an integer greater than zero. This indicates the highest outline level that will be initially opened when Acrobat displays the document. The default behavior is to have all levels open, but with exceptionally large reports, it may be desirable to have just the first 1 or 2 levels initially opened.

See the outline rule set in advanced.rul for an example.

Drivers: pdf only

OUTPUT

Syntax

1. output "*output-device*"

2. output {*expression*}

Description

The **output** keyword is used to modify the output device of any copy. Normally, all copies are printed to the output device specified in the "-o" option, or to standard out on UNIX. However, it is sometimes desirable to have copies of forms sent to different devices, such as a different laser printer, or a fax product.

The *output-device* can be a printer device, a pipe or re-direct (starting with | or >), or a filename. Beware of pipes or redirects on UNIX, noting that any shell-aware characters, such as ampersands (&), must be quoted.

If the second syntax is used, *expression* is evaluated after each page of input has been loaded and the prepage subroutine has been executed.

When used inside an **if copy** block, the output for that copy only is changed. Note that this feature is only supported in the pcl and postscript drivers. When using the pdf driver, any change to output for different copies is ignored.

The "output$" variable can also be set in a code block for equivalent results.

Example:

```
if copy 2
output "|lp -daccounting -s"
end if
```

The above example would send the second copy of the form to the printer named "accounting".

Drivers: laser, ps; pdf only for a job-wide specification outside of "if copy" blocks as PDF output cannot be changed during printing.

PAGE

Syntax

1. page *rows*

2. page *cols*, *rows*

Description

Syntax 1 specifies an input page length of no more than *rows* lines. If a form-feed character is encountered first, then the page is considered complete also. This keyword is useful if the application creates a form with line-feeds rather than form-feeds.

If syntax 2 is used, then each page worth of rows is divided into column groups of *cols* wide and treated as virtual pages from left to right. For example, if an application prints mailing labels as 4-up labels each 30 columns wide and 6 rows deep, then the command **page 30,6** would produce 4 pages, each 6 rows. This can be useful to convert *n*-up continuous label print jobs into laser label jobs using the **across** and **down** commands.

If no **rows** or **lpi** keyword is specified, then *n* is assumed to be the rows per page.

Examples:

page 42 treats each 42 lines of input as a full page.

page 42
rows 66 treats each 42 lines input as a full page, but produces output scaled to 66 lines per page.

Drivers: all

PostScript input not supported.

PAPER

Syntax

paper *size*

Description

The **paper** keyword overrides the "-paper" command line option. It tells UnForm the paper size to instruct the printer to use, and also defines the page size from which UnForm calculates column and row widths.

Common sizes for laser and PDF output are defined in ufparam.txt in the [paper] section. Such sizes include:

Value	**Size**
Letter	8.5 x 11 inches
Legal	8.5 x 14 inches
Ledger	11 x 17 inches
Executive	7.25 x 10.5 inches
A4	210 x 297 mm
A3	297 x 420 mm

In addition, you can specify the size in a format *width*x*height*. This feature defaults to inches and also supports specification in centimeters or millimeters, using a "cm" or "mm" suffix, such as **paper 20x30cm**.

If you specify the "custom" paper size for laser output, UnForm will use the defined size for scaling and will issue the proper custom paper command to the printer, but you may still have to modify the custom paper setting via the printer's control panel to avoid prompts to load custom paper into the printer.

Drivers: all

PORTRAIT, RPORTRAIT

Syntax

portrait or rportrait

Description

This keyword ensures that UnForm will print pages oriented in portrait (vertical) fashion. If, while analyzing the report text, UnForm detects a PCL control sequence to turn on landscape mode, then landscape will be the default orientation. Use this keyword to guarantee that the orientation will be vertical.

The **rportrait** command turns on reverse portrait mode.

Note that **portrait** is supported inside **if copy** blocks, allowing different copies to be in different orientations.

See also the **landscape** keyword.

Drivers: laser, ps, pdf (rportrait is laser only)

PRECOPY, PREDEVICE, PREJOB, PREPAGE

POSTCOPY, POSTDEVICE, POSTJOB, POSTPAGE

Syntax

precopy | postcopy | prejob | postjob | prepage | postpage {
code block
}

Note: the opening brace "{" needs to be on the same line as the keyword. The closing brace may follow the last statement, or be on the line below the last statement.

Description

These keywords are used to add Business Basic processing code to the form or report. They represent six different subroutines that UnForm executes at specific points during processing. The *code block* can be an arbitrary number of Business Basic statements; the total number of statements in all code blocks can be about 6,000.

- **prejob** executes after the rule set has been read, and after the first page is read, but before any printing takes place. Use this code to open files, define string templates, create user-defined functions, and initialize job variables.
- **postjob** executes after the last page has been printed. Use this to close out your logic, such as adding totals to log reports. There is no need to close files, since UnForm will RELEASE Business Basic.
- **predevice** executes just after a device has been opened. With the laser driver, the output device can be changed with the **output** command or by modifying the output$ variable in a prepage or precopy code block. Whenever a new device is opened for any given copy, this code block is executed. The programmer can then store information from the page that causes the device to be opened, such as a customer code or fax information.
- **postdevice** executes just after the output device has been closed. Use this code block to perform processing with prior output device, once UnForm has closed the device. For example, if the output device changed when the customer number changed, then one or more pages for a given customer would be in the output file and could be sent as a group to a fax product.
- **prepage** executes after each page is read, but before any printing takes place. Use this to gather data associated with any page, or to modify the content of the text if you need such modifications to apply to all copies.
- **postpage** executes after the last copy of each page has printed.
- **precopy** executes before each copy is printed. Use this to modify copy text content, to skip specific copies, or to modify a copy's output device.

- **postcopy** executes after each copy is printed.

Any valid Business Basic programming code can be entered, including I/O logic, loops, variable assignments, and more. Program to your heart's content. UnForm will add extensive error handling code within your code, and report syntax errors to the error log file or a trailer page.

Note that the **merge** command, while not executable code, is honored within a code block. The merged data must be valid code block syntax.

For more details about programming code blocks, see the Programming Code Blocks chapter.

Example:

This example shows how to use various routines to make copy 2 of a form be a conditionally faxed invoice, using a CSV formatted file containing a customer ID and a fax number.

```
prejob {
exportfile$=”/exports/faxnums.csv”
today$=dte(0:”YYYY-MM-DD”)
faxlog$=”/exports/logs/fax”+today$+.log”
}

prepage {
invoice$=get(65,5,7)
custid$=get(65,4,6)
custname$=trim(get(10,10,35))
faxnum$=getfilefield(exportfile$, custid$, 2)
}

precopy {
if copy=2 then:
        if faxnum$>"" then:
        output$="|fx -n "+faxnum$
        log(invoice$+"  "+custid$+" "+custname$, faxlog$)
        end if
end if
}
```

Drivers: all, but predevice and postdevice are only supported by laser and pdf drivers.

PROTECT

Syntax

protect [print] [,annotate] [,extract] [,modify] [,password “*password*” | {expr}] [, owner “*password*” | {expr}]

Description

Without the **protect** command, UnForm generates a standard PDF document that can be opened, viewed, printed, and modified by a user. This suffices for most business documents, but if an application requires protection of the PDF contents, then this command can be used. It adds encryption and protection to a PDF document.

By default, only viewing access is provided to users. Additional access can be granted by including the following options:

print adds the ability to print the document.

annotate adds the ability to add text annotations and fill in form fields.

extract adds the ability to copy text or graphics from the document for pasting into other applications.

modify adds the ability to modify document contents.

password "xxx"|{expr} sets user password required for opening document

owner "xxx"|{expr} sets owner password. If the owner password is used to open a document, Acrobat will allow modification to document permissions. Note an owner password won't automatically enable restricted options. Instead, it only allows changing of those permissions and saving of those new settings.

Password and owner expressions are interpreted at the start of the job, immediately after the prejob code block is executed; therefore only page one data, or variables defined in the prejob code block, are available for use.

Drivers: PDF only

ROWS

Syntax

rows *n*

Description

This keyword specifies the number of output rows to use for the form or report. The placement of each line is calculated to accommodate this many rows within the printable area of the paper. For example, with letter paper, the printable area is about 10.5 inches; **rows** 66 will cause each line to be 10.5/66 inches high. If present, this value will override any calculation based on the **lpi** keyword.

The number of rows (*n)* can be any value up to 255. It will default to 66 if no **rows**, **lpi**, or **page** keywords are present.

Note there is an important distinction between the **page** and **rows** commands. **Rows** refers to output scaling, whereas **page** defines the number of text lines to read per page from the input stream. However, if a **page** command is used, and a **rows** command is not, then the **rows** defaults to the value of the **page** command.

Examples:

rows 80 will set the line height to accommodate 80 rows per page.

Drivers: all

SHADE, CSHADE

Syntax

1. shade *col*|{*numexpr*}, *row*|{*numexpr*}, *cols*|{*numexpr*}, *rows*|{*numexpr*}, *percent* [,extend] [,*color*] [,rgb *rrggbb*]

2. shade *col*|{*numexpr*}, *row*|{*numexpr*}, *cols*|{*numexpr*}, *rows*|{*numexpr*}, *percent*, *skip*, *times* [,extend] [,*color*] [,rgb *rrggbb*]

3. shade "*text*|!=*text*|~*regexp*|!~*regexp*[@*left,top,right.bottom*]", *col*|{*numexpr*}, *row*|{*numexpr*}, *cols*|{*numexpr*}, *rows*|{*numexpr*}, *percent* [,extend] [,*color*] [,rgb *rrggbb*]

If cshade is used, then *cols* and *rows* are interpreted to be the opposite corner of the shade region, and columns and rows are calculated by UnForm.

Description

The region indicated by *col*, *row*, *cols*, and *rows* will be shaded, using the *percent* as the percent-gray value. The region parameters can be specified as decimal values to 1/100 character. The region is based on the full character cell, starting at the upper left corner of the cell. This differs from the **box** keyword, which measures from the center point of a cell. The *percent* can be any value from 0 to 100, where 0 is white (useful for erasing regions), and 100 is black. The default shade value is 5% (which renders as 10% in PCL5 devices). PCL5 printers actually support only eight levels of gray, generally: 2%, 10%, 20%, 35%, 55%, 80%, 99%, and 100%. Values less than these are rounded up to the next supported value. If you wish to issue a shade command that will do nothing, use -1 as the percent.

For compatibility with Version 1 rule files, Version 2 and above will convert shade values of 1, 2, 3, and 4 to 2%, 20%, 55%, and 100%, respectively.

If used, *numexpr* is a Business Basic expression that generates a numeric value for the column, row, columns, or rows.

Syntax 2 provides for repeating regions to be easily specified. The *skip* parameter is a number indicating the number of blank lines that follow the shade region. The *times* parameter is the number of times to repeat the shade/blank pattern. UnForm will generate multiple rows of shading until either the number of repetitions is met or the end of the page is found. For example, **shade 1,21,80,2,1,2,8** would produce 8 shaded regions, each 80 columns by 2 rows with shade grade level 1. Two blank lines would separate the shade regions. These two parameters are ignored if the first parameter is a text string, as in syntax 3.

If syntax 3 is used, then the shading is drawn relative to any occurrence of the *text*, or of text that matches the regular expression *regexpr*. In these cases, there may be no shaded regions, or several. *column* and *row* are 0-based, in these formats, and can be negative if required. The search for *text* or *regexpr* can be limited to a region on the page by adding a suffix in the format '*@left,top,right,bottom*'. To use a literal "@" character in *text* or *regexpr*, it is necessary to specify "\@".

If the syntax "!=*text*" or "!~*regexpr*" is used, then the search is for positions NOT equal to the text or NOT matching the regular expression. When using the NOT syntax, only one search is performed per line in the search region.

All formats support the **extend** option. This simply expands the shade region by ½ character in all directions, making it easy to fill in a box that is placed at the mid-point of each character position surrounding the shade region.

Note that the **box** keyword also supports shading, and may be more convenient to use if an outlined shaded region is desired.

Color can be specified as white, cyan, magenta, yellow, blue, green, red, or black, or you can name a RGB value as a 6-character hex string with rgb *rrggbb*, where *rr* is red (00-FF), *gg* is green (00-FF), and *bb* is blue (00-FF).

You can improve the look of shade regions on laser printers, especially at medium shade levels and 600 or higher dpi settings, by using the gs command.

With PostScript input, use a shade percent of 0 to erase a region of the overlay and allow UnForm graphical enhancements to remain visible.

Examples:

shade 41,3,40,6,2 will fill the indicated region with a medium (20%) shade.

shade 10.5,3.01,40,4.98,25 will shade the indicated region with 25% gray.

shade "No. Item/Desc",0,0,79,1,10,extend will shade from the position the noted text is found, for 79 columns and 1 line. The shaded region will then be extended ½ column and row in each direction. 10% gray will be used.

shade 1,14,80,2,1,2,12 will produce a repeated pattern of 80 columns wide, 2 lines high, light shading, followed by two blank lines. The pattern will be repeated 12 times, occupying a total of 48 lines.

Drivers: laser, pdf, ps

For Zebra (0% or 100% only), use a box command.

SHIFT

Syntax

shift *n*

Description

The text in the report is shifted *n* characters to the right (or left, if *n* is negative). If a report starts in column 1, but doesn't extend all the way to the right edge of the page, it is possible to shift the data to the right to allow for box drawing around text elements on the left margin.

The placement of relative shading, drawing, and attributes is determined *before* any shift.

See **vshift** also, for shifting text vertically.

Example:

shift 1 will shift all text 1 character to the right.

Drivers: all

PostScript input not supported.

SUBJECT

Syntax

subject "*subjectstring*" | {*expression*}

Description

If this command is present, then PDF document creation adds a subject *subjectstring*, or the result of *expression*, to the document content. This value is available in the general properties display dialog in the Adobe Acrobat Reader.

Drivers: PDF only

SYMSET

Syntax

symset "*symbolset"*

Description

The **symset** keyword overrides the default symbol set setting found in the [defaults] section of the ufparam.txt file. If there is no [defaults] section, then the symbol set 10U is used. Symbol set values for the LaserJet are always integers followed by an uppercase letter. Be sure to quote the *symbolset* value to maintain the uppercase letter (unquoted values in rule sets get converted to lowercase by UnForm's rule file parser).

Symbol sets are used to display specific international character sets or symbols. See your LaserJet documentation for symbol set codes supported by your printer.

If you plan to use the pdf driver in addition to the laser driver, you should specify your symbol sets as 9J if you intend to use special characters in the ASCII 128 to 255 ranges.

Drivers: laser only

TEXT

Syntax

1. text *col*|{*numexpr*}, *row*|{*numexpr*}, "*text*" | *@name* | *$name* | {*expression*} [,*fontname*] [,font *fontcode*] [,symset *symset*] [,*size*] [,bold] [,italic] [,underline] [,light] [,shade *percent*] [,rotate *angle*] [,fixed | proportional | prop] [,*color*] [,rgb *rrggbb*] [,*justification*, cols *ncols*|icols ncols|ccols *endcol*] [,wrap] [,fit] [,spacing *spacing*] [,weight *w*|*weightname*] [,style *style*|*stylename*]

2. text "*text*|!=*text*|~*regexp*|!~*regexp*[@*left,top,right.bottom*]", *col*|{*numexpr*}, *row*|{*numexpr*}, { "*text*" | *@name* | *$name* | {*expression*} } [,*fontname*] [,font *fontcode*] [,symset *symset*] [,*size*] [,bold] [,italic] [,underline] [,light] [,shade *percent*] [,rotate *angle*] [,fixed | proportional | prop] [,*color*] [,rgb *rrggbb*] [,*justification*] [,cols *ncols*|icols ncols|ccols *endcol*], [eraseoffset *cols*, erasecols *cols*] [,getoffset *cols*, getcols *cols*] [,wrap] [,fit] [,spacing *spacing*] [,weight *w*|*weightname*] [,style *style*|*stylename*]

Description

The *text* indicated in quotes will be printed at the column and row indicated by *col* and *row*. The column and row can be specified to 1/100 character. The position specified becomes the baseline left edge for the first character. If used, *numexpr* is a Business Basic expression that generates a numeric value for the column, row, columns, or rows.

If *text* begins with "@", such as **@company**, then the substitution file is searched. In the example above, if a line **company=ABC Company** was found, the text "ABC Company" is used. The substitution file defaults to "subst", but may be specified on the command line with the "-s" option.

If *text* begins with "$", then the operating system environment is searched for the indicated variable and its value is used. For example, **$USER** would use the value stored in the environment variable "USER".

If *text* should be a literal value that starts with @ or $, then use \@ or \$, respectively.

If braces surround *text*, then it is taken to be an expression to be evaluated after each page of input has been loaded and the **prepage** subroutine has been executed. The expression can be any valid Business Basic statement that would appear on the right side of an assignment statement and produces a string data type result. Some UnForm supplied functions and data can be useful, such as TEXT$[], which contains the text of the page in an array, and GET(col,row,length), a function that returns data from the TEXT$ array. For example, {"Copy 2, generated on "+date(0)} would generate text similar to this: "Copy 2, generated on 03/31/99". See the Programming Code Blocks chapter for more information about programming expressions.

If *text* contains line-feed characters (CHR(10) or $0A$), or the mnemonic character string "\n", then UnForm will break the text into multiple lines and space them according to the *spacing* value. For example, if the point size is 12, and *spacing* is set to 1.5, then line spacing is set to 18 points. The default *spacing* is calculated from the number of rows per page, so multi-line text data will match the vertical placement of single line text data.

If syntax 2 is used, then UnForm will search for occurrences of *text* or the regular expression *regexpr*. In this case, *col* and *row* become 0-based offsets from each location where matches are found. In addition, the erasecols *cols* and eraseoffset *cols* can be used to remove match text. The search for *text* or *regexpr* can be limited to a region on the page by adding a suffix in the format '*@left,top,right,bottom*'. To use a literal "@" character in *text* or *regexpr*, it is necessary to specify "\@".

If the syntax "!=*text*" or "!~*regexpr*" is used, then the search is for positions NOT equal to the text or NOT matching the regular expression. When using the NOT syntax, only one search is performed per line in the search region.

Font Names and Numbers
fontname can be Courier (the default), CGtimes, or Univers. These fonts are standard on virtually all PCL5 compatible printers. Alternately, a specific *fontcode* supported by your printer can be specified by its font number. For example, if your printer supports True Type Arial, specify "font 16602". Bitmap fonts (as opposed to scalable fonts) may be specified, but proper use depends on the form's or report's cpi value matching that of the font. Bitmap fonts have low *fontcode* values, like 0 for Line Printer, or 4 for Helvetica. *fontname* and *fontcode* values can also be specified from the "ufparam.txt" file.

Note that font 15002 is configured by default (in ufparam.txt) as a reference to the default MICR soft font, and can be used (with 'fixed, 8' options) to print MICR encoded text lines in cases where the micr command can't be used, such as with deposit slips, unusual bank account numbers, or non-standard check sizes.

Symbol Sets
symset can be any symbol set supported by your printer. The default symbol set is "9J", using the Windows Latin 1 character set (similar to ISO8859-1, but with some additional characters defined). You can also specify symbol sets by name from the "ufparam.txt" file. Only symbol set 9J is supported by the PDF and Postscript drivers.

To include non-printable characters, such as control codes or 8-bit characters from a specific symbol set, include the character's numeric (ASCII) value in angle brackets. For example, to include a copyright symbol from the Desktop (7J) symbol set, use something like this: "<165>2000 Synergetic Data Systems Inc.", or use an expression.

Point and Pitch Size
size is a numerical value that specifies the point size of a proportionally spaced font or the pitch size of a fixed font. The values range from about 4 to 999.75 with a default of 12. PCL printers generally round this value to the nearest or smallest ¼ point. Note that for proportional fonts, the larger the number, the larger the size printed. Fixed fonts, such as Courier, are the opposite. If you specify the "fit" option, then the *size* value represents the largest acceptable size.

Fit and Wrap Options
The "fit" option will scan *text* for line breaks and decrease the *size* value as necessary to ensure that all lines will fit in the number of specified *ncols* or through *endcol*. The smallest point size that will be used is 4, and the largest pitch that will be used is 30.

The "wrap" option will scan *text* and insert line breaks as needed to ensure no line at the specified *size* will exceed the specified *ncols*. If no spaces exist in word that exceeds the line width, UnForm will print the word in its entirety, exceeding the allocated space.

The "fit" and "wrap" options are mutually exclusive, and in either case, if no *ncols* or *endcol* value is specified with the "cols" option, then *ncols* defaults to the page width in columns minus *column*.

Attribute Styles
The attribute words "bold", "italic", "underline", and "light" will apply the indicated attribute(s) to the text.

Shading
percent indicates the percent gray to print the text, from 0 (white) to 100 (black). The default is black. Note that the **gs** command can be used to improve laser printer shading.

Rotation
The "rotate" option will cause the text to be rotated around the baseline left edge at the angle specified. If the –nohpgl option is used while producing pcl laser output, then the angle must be 90, 180, or 270 degrees. Any angle can be specified for other formats.

Fixed and Proportional Spacing
Specify "fixed" or "proportional" (or "prop") to override the default of fixed for Courier (or any *fontcode* below 4100), and proportional for all else. For example, if a mono-spaced font, such as the MICR soft font, has a font code higher than 4100, then the "fixed" option is required in order to ensure the proper font is selected, rather than a default proportional font. Proportional vs. fixed carries a very high priority when a printer chooses a font, and if the desired font is not specified with the correct spacing, a different font will be chosen by the printer.

Color
Color can be specified as "white", "cyan", "magenta", "yellow", "blue", "green", "red", or "black", or you can name an RGB value as a 6-character hex string with "rgb *rrggbb*", where *rr* is red (00-FF), *gg* is green (00-FF), and *bb* is blue (00-FF).

Justification
justification can be one of the following words: "left", "center", "right", "decimal". UnForm will remove leading and trailing spaces from the text and justify it within the column specification. Decimal justification will use a "." character unless a "decimal=*character*" line is placed in the ufparam.txt file under the [defaults] section.

For justification, you must also specify *ncols* or *endcol* with the "cols", "icols", or "ccols" option, so that UnForm can determine the right edge of the justification region.

Weight and Style
Some laser printer fonts must be specified with given weight or style in order to be selected by the printer. For example, the font Clarendon Condensed is only available if the condensed style is specified, by adding "style 4" or "condensed" to the font command. Style and weight options and codes can be found in the ufparam.txt file. Note that fonts are expressly designed for certain weights and styles, and simply specifying an unsupported value does not produce the desired result. In fact, it may result in selection of a different font entirely. Check your printer's documentation or control panel prints for supported fonts.

Get Text From Input Stream
If "getoffset" and "getcols" are specified in a syntax 2 command, then the value printed is taken from the data stream at the offset and length specified from each occurrence (any *text* value supplied is ignored). Further, "eraseoffset" and "erasecols" can be used to remove any data stream text from the point of occurrence as well.

Barcode Note
The text command can be used to print a human-readable version of a barcode value, which can be useful in cases where the human readable value differs from the supplied value, such as UPC-E, or when a check digit value is needed.

Text in this syntax: "bcd*sss*|*value*" to print the human readable barcode value for symbology *sss* and barcode text *value*, "ck1*sss*|*value*" to print check digit 1, or "ck2*sss*|*value*" to print check digit 2. See the barcode command for symbology values.

Special Symbol Fonts
There is a difference between PDF and laser output for special symbols. In the laser printer environment, you need to select a symbol set *and* font that contains the special characters you want, but in the PDF environment, you need only select the font (font 4141 for Dingbat and 16686 for Symbol). Once a symbol set or font is identified, use the appropriate decimal value of text to print the character you want. The easiest way to do this is with angle bracket notation in a literal, like "<182>", or with the CHR function in an expression, like {CHR(182)}.

On many LaserJet printers, the available symbol sets and fonts differ from those specified in UnForm's ufparam.txt file, and the only way to know for sure what is available is to do a font list print on the printer. This should show you the proper symbol set and font number to use for your printer.

Postscript Type1 Fonts
All Postscript printers include a base set of fonts, including Courier, Times-Roman, and Helvetica. The [psmap] section of ufparam.txt links font numbers to Postscript font names, and the most often used font numbers are mapped in the default ufparam.txt file. The [psmap] section looks like this:

```
[psmap]
# Maps pcl font numbers to postscript font names.  Each number is
# set to up to four font filenames for normal, bold, italic, and
# bold-italic fonts in order.  If a fontname.pfa or fontname.pfb
# file is present in the psfont directory, then it is downloaded when used.
# Such fonts must be Adobe Type1 fonts.
#
# psfont/<fontname>.afm files provide metrics.

4099=Courier,Courier-Bold,Courier-Oblique,Courier-BoldOblique
4101=Times-Roman,Times-Bold,Times-Italic,Times-BoldItalic
4102=Courier,Courier-Bold,Courier-Oblique,Courier-BoldOblique
4141=ZapfDingbats
```

In order to use custom Type1 fonts, follow these steps:

- Install the font’s .pfa or .pfb files, as well as the metric file (.afm) in the psfont directory.
- Assign a unique number to the fonts, using the same normal, bold, italic, and bold-italic sequence found in the standard font mapping, if the font provides these versions.
- Optionally map a font name to the number in the [fonts] section of ufparam.txt.
- Text or font commands can now specify the font by number (font *n*) or name.

TrueType Fonts and Unicode

TrueType fonts can be configured by assigning font numbers to font file names in the [ttmap] section of ufparam.txt, in a manner similar to the [psmap] section described above. In the Windows environment, the system's Fonts directory is automatically searched as well as the ttfont directory under the UnForm server, so there is no need to define full paths to system fonts on Windows.

When using a TrueType font, text data must be in Unicode format, where each character is represented with two bytes rather than one. Standard single-byte encodings therefore need to be converted to Unicode, by mapping the characters in the string to the proper double-byte characters in Unicode. Most character sets use the same characters in the 0-127 character range, but vary in the 128-255 range. Unicode can represent any of these characters with a specific 0-65535 value, allowing any character set to be represented in Unicode. The tables for most PCL symbol sets as well as several common character sets are included with UnForm in its unicode directory, and UnForm provides internal and code block functions to perform mapping.

Normally, UnForm will assume that the text data provided to the command is single-byte data, and it will internally map the text to a Unicode string based on the symbol set setting. The default symbol set (9J) matches the ISO-8859-1 character set, so that is the default translation.

If you wish to specify Unicode directly, you can do so in a variety of ways. You must add a "uc" or "unicode" option to the text command to indicate the text data is already in Unicode format.

- Use the touc(text$,charset$) function in an expression, which returns the Unicode version of text$, given that text$ is in the identified character set or symbol set. For example, {touc("Total Sales","9J")}.
- Specify raw double-byte sequences as hexadecimal in an expression, using $*hex*$ notation, such as {00300039}.
- Use the <x*hex*> syntax in literal text, such as "<x00300039>".

Examples:

text 10,2,"SOLD TO" prints the text SOLD TO at the indicated position.

text 120,3,$LOGNAME prints user's login name at column 120, line 3.

text 1.25,63.25,{"Printed on "+date(0)}, cgtimes, 6, italic would place a small (6 point), italic note about the date near the lower left corner of a page.

text "TOTAL:",-1,0,"Total:",cgtimes,12,bold,eraseoffset 0, erasecols 6 changes words TOTAL: to Total: in CGTimes, 12 point, after backing up 1 column from where TOTAL: is found. It also erases the word TOTAL: to avoid overprinting.

text 67,21,"bcd125|00010000654",univers,12 will print the UPC-E human readable barcode value.

text 20,62,{terms$},cgtimes,10,cols 40,wrap,spacing 1 will print a paragraph of text contained in terms$ between column 20 and 59, in CGtimes 10 point text, word-wrapping as necessary, using a nominal line height matching the 10 point text.

text {pos("Item"=text$[20])},21,"Number",cgtimes,12 will print the word "Number" on line 21, in the same column where the word "Item" is found in line 20.

text 20.5,20,{mcut(10,20,12,40,"","y","y")},cgtimes,12,right will cut text from the data stream, at column 10, row 20, for 12 columns, 40 rows, retaining line breaks, and print it as a column of 40 rows at column 20.5, row 20. The column will be printed in the font CGtimes, 12 point size, right justified.

text 1,60,{mcut(1,60,200,5,"","","")},univers,10,wrap,cols 60 will cut a large message block from the data stream, at column 1, row 60, for 200 columns, 5 rows, removing line breaks. It will then print it at column 1, row 60, at 10 point size and word wrapping to make it fit within 60 columns.

Drivers: all. PDF driver fonts map to Courier, Helvetica, or Times-Roman based on the [pdfmap] section if ufparam.txt, and support only symbol set 9J (ISO8859-1 with extensions). PS maps based on the [psmap] section in ufparam.txt. Zebra fonts are limited in scalability, and the font codes are letters or numbers that identify internal font codes specified in the ZPL documentation. Zebra shading is limited to 0% or 100%. Zebra doesn't support colors or decimal justification, or wrap and fit options. **Light** and **underline** options are only supported by the pcl driver.

TITLE

Syntax

title "*titlestring*" | {*expression*}

Description

If this command is present, then PDF document creation adds a title *titlestring*, or the result of *expression*, to the document content. This value is available in the general properties display dialog in the Adobe Acrobat Reader.

The title command is also honored by the UnForm Desktop Client, though expressions are not supported.

Drivers: pdf only

TRANSPARENCY

Syntax

transparency on|yes|off|no

Description

The transparency command turns on or off PDF transparency mode, which affects shading created by the image, text, box, and shade commands. When transparency mode is on, shading is implemented as a partial transparency, which allows objects to show through the shade region.

Transparency can also be controlled in the uf80d.ini file (pdftrans=*n*) or with the -pdftrans or -nopdftrans command line options.

Drivers: PDF only

TRAY

Syntax

tray *paper-source*

Description

The **tray** keyword can be used to specify the paper source for any copy or for the print job. If, for example, you have two input trays, one with letterhead stock and one with plain stock, you can specify which paper stock to use for any form or copy of a form.

Trays can be varied by copy, to pull different paper for different copies.

The *paper-source* is printer dependent. Typically, tray 1 is an upper tray source, tray 2 is a manual feed source, and tray 4 and 5 are a lower and middle paper sources. These will likely not coincide with physical tray numbers labeled on the printer itself, unfortunately. To determine the proper tray values, see your printer's documentation for the paper source command.

The printer model's (-m command line option) PPD file (or generic pcl.ppd or ps.ppd files) can specify *InputSlot *paper-source* entries which are used if present.

When producing output to a *winprt* Windows printer, the tray command specifieds a tray number that is assigned by the vendor, typically a number over 256, though there are pre-defined Windows values published by Microsoft that some vendors honor. A list of tray numbers for a Windows printer can be obtained with the system object, using the winprttrays$(printer$) method.

Example

```
tray 5

if copy 2
   tray 4
end if
```

Drivers: laser, ps, *winprt* only

UNDERLINE

See the **bold** keyword.

UNITS

Syntax

units dpi | char

Description

As UnForm parses a rule set, column and row specifications are normally interpreted as decimal column and row numbers that align enhancement elements such as boxes and shade regions with characters in the source data. If you need to specify absolute dot positions, however, you can change the units to dpi. From that point in the rule set, until a **units char** is found, row and column values are interpreted as integer dot positions. Note that the **dpi** keyword has a direct impact on dpi units, though no impact on char units.

For example, the following will print two text phrases at column 1 inch, row 1.5 inch.

```
units dpi
text 300,450,"Hello, world"
dpi 600
text 600,900,"Over printing hello world"
units char
```

Drivers: laser, ps, PDF

VLINE

Syntax

vline "*text*" [,erase] [,extend] [,*thickness*]

Description

Any vertical occurrence of the *text* indicated, of at least the length indicated, will be replaced with a vertical line. The *text* must be composed of a single character repeated any number of times. There can be multiple **vline** keywords in a rule set, if needed.

This keyword is useful if the application already produces boxes and lines with standard characters. See also the **hline** keyword.

As with all box drawing, UnForm will consider line end-points to be at the center position of a character, which may impact how lines intersect. Lines are drawn one dot (1/300th inch) thick.

If the "erase" option is used, then no line is drawn. Instead, the vertical text values are simply removed from the output.

If the "extend" option is used, the lines are extended ½ characters up and down. The *thickness* parameter specifies a pixel width to draw.

The search for *text* can be limited to a region on the page by adding a suffix in the format '*@left,top,right,bottom*'. To use a literal "@" character in *text*, it is necessary to specify "\@".

Example:

vline "|" will search the report for pipe characters. All such characters found will be replaced with vertical line draw (box) characters.

Drivers: all

PostScript input not supported.

VSHIFT

Syntax

vshift *n*

Description

The **vshift** keyword shifts text vertically down (or up, if *n* is negative) the indicated number of lines. The shifting is done before placement of any fixed shading or boxes. Lines shifted out of the printing region (line 1 through the page specification, or 255 if not specified) are not printed. See the **shift** keyword, also, for horizontal shifting.

The placement of relative shading, drawing, and attributes is determined *before* any shift.

Example:

vshift 1 shifts all text down 1 line, providing room for a box definition at the top of the page.

Drivers: all

PostScript input not supported.

ZCOPIES, ZDARKNESS, ZSPEED

See the **lcopies**, **ldarkness**, and **lspeed** commands.

WORKING WITH MACROS

Using macros can increase the speed and efficiency of printing your enhanced forms and documents by storing fixed raster graphics (e.g. logos) on the printer instead of transmitting these graphics on every page being printed. With the graphics stored on the printer, only 12 to 14 bytes are transmitted to the printer to select the macro to print. The time savings for printing are most noticeable when your system can't communicate to your printer at a high speed. For parallel or local network connections, macro usage doesn't often make too much difference. However, if you use serial connections or wide area network printing with low- or shared-bandwidth, then implementing macros can help performance. The more graphics used in enhancing forms, the more print transmission time you can save by using macros.

The PCL5 specification defines two types of macros: temporary and permanent. Temporary macros are downloaded at the start of a print job, and can be executed by the printer until it is reset at the end of the job. Permanent macros remain in printer memory until the printer power is turned off. A number from 1 to 32767 always identifies macros.

To access permanent macros, simply add **macro *n*** (n=macro #) to the rule set. To instruct UnForm to utilize temporary macros, add the **macros on** command to the rule set. UnForm will then generate temporary macros for any fixed elements of the job, download them at the start of the job, and execute them as the job is printed.

If you print large batches of forms at one time, and use a serial or low-bandwidth network connection, temporary macros can produce considerable time savings by reducing the amount of data transmitted to the form. For example, if a logo image is 20,000 bytes, and line drawing and shading add another 5,000 bytes, a 50 page form will save about 49 x 25,000 bytes, or about 1.2MB. At typical serial throughput, this could save as much as 10 minutes of print time. High-speed printer connections (parallel or local network) only produce minimal time savings, which is sometimes offset by the extra overhead incurred by UnForm to manage the macros in memory.

UnForm also provides the ability to generate permanent macro files. Permanent macros can be downloaded when the printer is turned on, and then UnForm can execute them without the overhead of downloading them at the start of a job. To utilize this enhanced functionality, you must modify the rule file and create a command line script to load the graphics into the printer.

To use this capability, you should split a rule set into two rule sets. One will be used to generate the permanent macros (there can be a macro for each copy defined in the rule set); the other will be used as before, but will replace the elements placed in the macros with **macro *n*** commands.

The rule set used to generate the macro can contain these commands that are in fixed positions: **image**, **attach**, **box**, **shade**, and **text**. It can also contain **if copy** blocks. It should not contain any other commands or any of the named commands if they incorporate relative positioning. **Detect** commands are ignored; you will use the "-r *ruleset*" command line option instead. The remaining commands should be left in the original rule set, and **macro *n*** commands added based upon the macro numbers assigned in the command described below.

Next, you need to generate macro files for each copy that is used in the rule set. To do this, use this command line:

uf80c –makemacro *macro-number* –f *rulefile* –r *macro-rule-set* –macrocopy *copy* –o *output-file*

UnForm will generate a permanent macro in *output-file*, numbered as *macro-number*.   This is the same number you would then specify in the regular rule set, as macro *macro-number*.   On UNIX, the output can be piped directly to the spooler, either by removing the –o option or by using a quoted pipe as the output file: –o "|lp –o raw –d *printername*".

REGULAR EXPRESSIONS

Regular expressions are supported in many of UnForm's keywords, and can be used to great advantage in detect statements and relative enhancements. Regular expressions are similar to, but much more powerful than, MS-DOS or UNIX *wildcards*.

A regular expression is used to match patterns in text. By using special characters, called *meta characters*, UnForm can be instructed to search for patterns, such as dates or codes, and use them in processing. Below is a description of the various meta characters and how to use them.

- The simplest regular expression contains no meta characters. It just matches itself. **John** will match any occurrence of the text "John".
- Brackets can be used to match any of a group of values: **[Jj]ohn** will match both "John" and "john".
- If a range of letters or numbers is valid in a position, then the range can be indicated in a similar manner: **[A-Za-z]ohn** will match any letter, upper or lower case, followed by the letters "ohn".
- If single character positions are not enough, then groups of options can be used with parentheses and vertical bars, like this: **(John)|(Jack)|(Jill) Smith**, which matches any of the first names, along with "Smith".
- If any character will do in a position, use a dot: **Jo.n** will match "Jo", followed by any single character, followed by "n".
- To repeat any pattern, including a dot, use an asterisk (*) for 0 or more repetitions, or + for 1 or more repetitions: **J.*n** will match a "J", followed by 0 or more characters, followed by "n". **Jo+n** would match a "J" followed by one or more "o"s, followed by "n".
- You can include multiple meta characters and patterns in the expression. For example, to search for 3 digits followed by 2 letters: **[0-9][0-9][0-9][A-Z][A-Z]**.
- To disable the special meaning of any of the meta characters, prefix it with a backslash. For example, a phone number might include parentheses; to include them in the expression, they must be disabled: \\**(...\\)-...-....**.
- The meta characters are: ., *, +, (,), |, [,], ^, and $.

SAMPLE RULE FILES

UnForm is supplied with several sample report text files and associated rule sets. A description of each report and rule set follows. Each of the sample reports is in the UnForm directory, named "sample*n*.txt." All example rule sets can be found in the files simple.rul and advanced.rul in the UnForm directory.

The simple.rul file contains a series of examples that use the sample invoice text file, sample1.txt. Beginning with the rule set simple1, and incrementally advancing in capabilities through simple4, this rule file is designed to help a new user learn fundamental UnForm concepts. To try these out, use this command, varying the rule set name (-r argument) simple1 with one of the four samples, simple1, simple2, simple3, or simple4:

uf80c –i sample1.txt –f simple.rul –r simple1 –o *output-device*

The advanced.rul file contains rule sets that show a variety of topics, and is designed to show advanced concepts. To produce these samples on your own laser printer or to a PDF file, you can use the following command, substituting the proper sample text file:

uf80c –i *sample-file* –f advanced.rul –o *output-device*

For the output device, you can use a device name, like LPT1 or /dev/lp0, a file name, or a quoted pipe command to a spooler. For example, to print the first sample to a spooler, use something like this:

uf80c –i sample1.txt –f advanced.rul –o "|lp –dhp –oraw"

To produce PDF versions of these files, change the output device to a PDF file name, and add "-p pdf" to the command line:

uf80c –i sample1.txt –f advanced.rul –p pdf –o invoice_sample.pdf.

Change "-o invoice_sample.pdf" to "-o client:invoice_sample.pdf" to store the output on the client's system.

A few of the samples don't support detection capabilities, and they must be specified on the command line with a "-r *ruleset*" option. If necessary, the documentation will state this requirement.

SIMPLE1 - invoice rule set (simple.rul)

This is the first example of an invoice rule set, found in simple.rul. To produce this example:

uf80c –i sample1.txt –f simple.rul –p pdf –o client:simple1.pdf

A title header prefixes all rule sets, which is just a unique name enclosed in brackets.

```
[simple1]
```

Detect statements are used to identify this form from any other report that the application might send to the printer through UnForm. Unlike most form packages, UnForm doesn't dedicate a printer name to a particular form (though it can be configured to do so). Instead, it reads the first page of data, then compares it to the detect statements found in the various rule sets in the rule file.

The detect statements below indicate that

- *a date (mm/dd/yy format) followed by 2 spaces, followed by 7 more characters will appear at column 61, row 5*
- *6 characters will appear at column 9, row 11*
- *a date, a space, and 6 characters will appear at column 10, row 21*

```
detect 61,5,"~../../..  ......."        # invoice date and #
detect 9,11,"~......"                   # customer code
detect 10,21,"~../../.. ......"         # ord date and cust code
```

The following lines define that the dimensions of the page are 80 columns by 66 rows. All positioning will be based on 80 columns and 66 rows appearing within the printed margins of the page.

```
cols 80                                 # max output columns
rows 66                                 # max output rows
```

The header section draws a box around the entire form with a cbox command, then adds a logo and some header text. The" \n" character sequence represents a line break, so you can print a column of text easily. All the text commands are using the univers font, which is standard in all supported laser printers and which maps to Helvetica in PDF output.

```
# header section
cbox .5,.5,80.5,66.5,5
image 1,1,12,6,"sdsilogo.pcl"
text 15,2,"Company Name",univers,14,bold
text 15,3,"Company Address\nCompany City, St Zipcode\nCompany Phone",univers,12,bold
text 15,6,"Web: www.myweb.com\nEmail: sales@myweb.com",univers,11,bold
text 70,2,"INVOICE",univers,16,bold
```

The upper right of the form contains a box with grid lines and some title text, placed around the existing text supplied from the input stream (in this example, the file sample1.txt). The cbox command draws an outer box using the primary dimensions, and then adds internal horizontal lines at rows 6 and 8, and internal vertical lines at columns 69 and 78. The second row simply duplicates the bottom row, but adds 20% shading between rows 6 and 8.

Additional heading and box sections are drawn in a similar manner, for the remainder of the form. All the drawing simply adds details on top of, or around, the input data stream.

```
# invoice # section
cbox 60,4,80.5,8,crows=6 8::20,ccols=69 78
text 61,7,"Date",univers,italic,10
text 70,7,"Invoice #",univers,italic,10
text 79,7,"Pg",univers,italic,10

# bill to / ship to section
cbox .5,10,80.5,18.5,5,ccols=7::20 43.5 50::20
text 2,11,"Sold To",cgtimes,italic,10
text 45,11,"Ship To",cgtimes,italic,10

# ribbon section
cbox .5,18.5,80.5,22.5,5,crows=20.5::20,ccols=9 18 25 65
# special internal grid in ribbon box
cbox 29,18.5,65,21.5
cbox 42,18.5,56,21.5
text 1,19,"Order\nNumber",univers,italic,10
text 10,19,"Order\nDate",univers,italic,10
text 19,19,"Cust.\nNumber",univers,italic,10
text 26,19,"Sls\nPrs",univers,italic,10
text 30,19,"Purchase\nOrder No.",univers,italic,10
text 43,19,"\nShip Via",univers,italic,10
text 57,19,"Ship\nDate",univers,italic,10
text 66,19,"\nTerms",univers,10,italic

# detail section
cbox .5,22.5,80.5,56.5,5,crows=24.5::10,ccols=5 10 16 51 55 69
text 1,23,"Qty\nOrd",univers,italic,10
text 6,23,"Qty\nShip",univers,italic,10
text 11,23,"Qty\nBkord",univers,10,italic
text 17,23,"\nItem & Description",univers,italic,10
text 52,23,"\nU/M",univers,italic,10
text 56,23,"Unit\nPrice",univers,italic,10
text 70,23,"Extended\nPrice",univers,italic,10

# footer section
cbox 57,57,80.5,65,crows=59 63,ccols=69::20
text 58,58,"Sales Amt",univers,11
text 58,61,"Sales Tax",univers,11
text 58,62,"Freight",univers,11
text 58,64.25,"TOTAL",univers,bold,14
```

SIMPLE2 – invoice rule set (simple.rul)

This is a somewhat more advanced rule set than simple1, demonstrating how to add fonting, justification, and text movement to the job. Additional notes are supplied to highlight these concepts. To prevent simple1's detection code from selecting the job, add a –r option to the command line:

uf80c –i sample1.txt –f simple.rul –r simple2 –p pdf –o client:simple2.pdf

```
[simple2]
# to use this rule set, you need to FORCE the rule set with the -r option
# or remark (#) out the detect command in the rule sets above.
#
# This rule set takes the rule set above and improves it by adding
# fonting and justification.

# It also cuts and pastes the invoice #/date/pg fields which allows
# more room for company name and address to be centered

# Also notice first use of relative expression in a text command to fix
# a problem with fonting a series of rows. Put a # in front of this
# command to see the problem that occurs. See memo section.
#

detect 61,5,"~../../..  ......."        # invoice date and #
detect 9,11,"~......"                   # customer code
detect 10,21,"~../../.. ......"         # ord date and cust code

cols 80                                 # max output columns
rows 66                                 # max output rows
```

The header section has changed to use center and right justification. Note the use of cols=79 *in each text command, which tells UnForm the bounds of the justification region. For example, the text "Company Name" is centered in the region starting at column 1, for 79 columns.*

```
# header section
cbox .5,.5,80.5,66.5,5
image 1,1,12,6,"sdsilogo.pcl"
text 1,2,"Company Name",univers,14,bold,center,cols=79
text 1,3,"Company Address\nCompany City, St Zipcode\nCompany
Phone",univers,12,bold,center,cols=79
text 1,6,"Web: www.myweb.com\nEmail: sales@myweb.com",univers,11,bold,center,cols=79
text 1,2,"INVOICE",univers,16,bold,right,cols=79
```

The Invoice number section is re-formatted here, by first drawing a new, vertically-oriented grid and heading section, then by using text commands with expressions that use the cut() function. The expression is indicated by the curly braces, such as {cut(61,5,8,"")}, *which directs UnForm to resolve the function as the job processes each page. In the line starting with* "text 75,5", *the data from the input stream at column 61, row 5, for 8 characters is cut and replaced with nothing (""), and it becomes the value printed at column 75, row 5.*

Further down, in the bill to/ship to section, is an example of the mcut() function, which cuts multiple lines and replaces them with "", retaining line breaks and trimming each line of leading and trailing spaces.

```
# invoice # section
cbox 67,4,80.5,10,1,crows=6 8,ccols=74::20
text 68,5,"Date",univers,italic,10
text 68,7,"Invoice",univers,italic,10
text 68,9,"Page #",univers,italic,10
# cut data from old position and place in new
text 75,5,{cut(61,5,8,"")},cgtimes,bold,10
text 75,7,{cut(71,5,7,"")},cgtimes,bold,10
text 75,9,{cut(79,5,2,"")},cgtimes,bold,10

# bill to / ship to section
cbox .5,10,80.5,18.5,5,ccols=7::20 43.5 50::20
text 2,12,"Sold To",cgtimes,italic,10,center,cols=5
cfont 8,11,40,11,cgtimes,bold,10,left
cfont 8,12,40,15,cgtimes,bold,10                          # sold to address
text 45,12,"Ship To",cgtimes,italic,10,center,cols=5
cfont 51,11,80,11,cgtimes,bold,10,left
text 51,12,{mcut(51,12,30,4,"","Y","Y")},cgtimes,bold,10

# ribbon section
cbox .5,18.5,80.5,22.5,5,crows=20.5::20,ccols=9 18 25 65
# special internal grid in ribbon box
cbox 29,18.5,65,21.5
cbox 42,18.5,56,21.5
text 1,19,"Order\nNumber",univers,italic,10,center,cols=8
text 10,19,"Order\nDate",univers,italic,10,center,cols=8
text 19,19,"Cust.\nNumber",univers,italic,10,center,cols=6
text 26,19,"Sls\nPrs",univers,italic,10,center,cols=3
text 30,19,"Purchase\nOrder No.",univers,italic,10,center,cols=12
text 43,19,"\nShip Via",univers,italic,10,center,cols=13
text 57,19,"Ship\nDate",univers,italic,10,center,cols=8
text 66,19,"\nTerms",univers,italic,10,center,cols=14
```

This section changes the fonts of the input data stream in the invoice ribbon section. For example, the first cfont command changes the data in column 1, row 21 through column 8, row 21, to cgtimes, bold, 10 point, centered text. Note how the font command applies to the incoming data stream, which differs from the text command, which adds additional output to the job. Font commands therefore work with integer positions, as they modify the character-base data stream as it passes through to the output.

```
cfont 1,21,8,21,cgtimes,bold,10,center      # order #
cfont 10,21,17,21,cgtimes,bold,10,center    # order date
cfont 19,21,24,21,cgtimes,bold,10,center    # cust #
cfont 26,21,28,21,cgtimes,bold,10,left      # sls prs code
cfont 26,22,64,22,cgtimes,bold,10,left      # sls prs name
cfont 30,21,41,21,cgtimes,bold,10,center    # po #
cfont 43,21,55,21,cgtimes,bold,10,center    # ship via
cfont 57,21,64,21,cgtimes,bold,10,center    # ship date
cfont 66,21,80,22,cgtimes,10,center         # terms

# detail section
cbox .5,22.5,80.5,56.5,5,crows=24.5::10,ccols=5 10 16 51 55 67
text 1,23,"Qty\nOrd",univers,italic,10,right,cols=4
text 6,23,"Qty\nShip",univers,italic,10,right,cols=4
text 11,23,"Qty\nBkord",univers,10,italic,right,cols=4
text 17,23,"\nItem & Description",univers,italic,10
text 52,23,"\nU/M",univers,italic,10,center,cols=3
text 56,23,"Unit\nPrice",univers,italic,10,right,cols=11
text 68,23,"Extended\nPrice",univers,italic,10,right,cols=12
```

This section performs two distinct fonting functions. First, the detail item columns are fonted. Note that you can't simply font the entire detail section in a proportional font, as the spacing between columns would be lost. Instead, each column is fonted individually.

However, the data stream for the invoice also contains memo lines in the middle of the detail item lines, and those memo lines should not be broken into individual columns.

Therefore, an additional font command is added after the column fonting, which will override any font characteristics defined for any given data position in a prior font command. This memo section fonting uses a technique that will scan the page for a pattern (in this example, 4 spaces in the region outlined by column 1, row 25 through column 4, row 56), and change font characteristics relative to those locations found. In this example, wherever the 4 spaces are found, fonting will occur 17 columns to the right, 0 rows down, for 60 columns and 1 row. These are the memo lines found in the midst of the item detail lines.

```
cfont 1,25,4,56,cgtimes,10,bold,right        # qty ord
cfont 6,25,9,56,cgtimes,10,bold,right        # qty shipped
cfont 11,25,15,56,cgtimes,10,bold,right      # qty b/o
cfont 17,25,50,56,cgtimes,10,left            # item # & desc
cfont 52,25,54,56,cgtimes,10,bold,center     # u/m
cfont 56,25,66,56,cgtimes,10,bold,right      # unit price
cfont 68,25,79,56,cgtimes,10,bold,right      # extended

# memo section
font "    @1,25,4,56",17,0,60,1,cgtimes,10,left

# footer section
cbox 57,57,80.5,65,crows=59 63,ccols=67::20
text 58,58,"Sales Amt",univers,11
cfont 58,60,66,60,univers,11,left
text 58,61,"Sales Tax",univers,11
text 58,62,"Freight",univers,11
text 58,64.25,"TOTAL",univers,bold,14
cfont 68,58,79,65,cgtimes,bold,right,14                  # totals
```

SIMPLE3 – invoice rule set (simple.rul)

This rule set adds copy handling, a watermark, and a barcode. To produce this sample, use this command:

uf80c –i sample1.txt –f simple.rul –r simple3 –p pdf –o client:simple3.pdf

A title header prefixes all rule sets, which is just a unique name enclosed in brackets.

```
[simple3]
# to use this rule set, you need to FORCE the rule set with the -r option
# or remark (#) out the detect command in the rule sets above.
#
# This rule set takes the rule set above and improves it by adding
# copies, watermark, and a barcode.

dsn_sample "sample1.txt"
detect 61,5,"~../../..  ......."        # invoice date and #
detect 9,11,"~......"                   # customer code
detect 10,21,"~../../.. ......"         # ord date and cust code

cols 80                                 # max output columns
rows 66                                 # max output rows
```

This rule set produces two copies of each page, with each copy sequentially produced as each page is read from the data stream. The pcopies command indicates this page-oriented copy production. There is also a copies command, which produces job-oriented copies for laser jobs. Note that PDF output always is produced as page-oriented copies, whether copies or pcopies is used.

When copies are produced, all rule set content that is not bracketed within 'if copy' blocks is produced on all copies. The majority of this rule set is outside of such blocks, so the majority will be applied to both copies. Near the bottom of the rule set is some code that will apply distinctly to each copy.

```
# copies
pcopies 2

# header section
cbox .5,.5,80.5,66.5,5
image 1,1,12,6,"sdsilogo.pcl"
text 1,2,"Company Name",univers,14,bold,center,cols=79
text 1,3,"Company Address\nCompany City, St Zipcode\nCompany
Phone",univers,12,bold,center,cols=79
text 1,6,"Web: www.myweb.com\nEmail: sales@myweb.com",univers,11,bold,center,cols=79
text 1,2,"INVOICE",univers,16,bold,right,cols=79

# invoice # section
cbox 67,4,80.5,10,1,crows=6 8,ccols=74::20
text 68,5,"Date",univers,italic,10
text 68,7,"Invoice",univers,italic,10
text 68,9,"Page #",univers,italic,10
# cut data from old position and place in new
```

```
text 75,5,{cut(61,5,8,"")},cgtimes,bold,10
text 75,7,{cut(71,5,7,"")},cgtimes,bold,10
text 75,9,{cut(79,5,2,"")},cgtimes,bold,10

# bill to / ship to section
cbox .5,10,80.5,18.5,5,ccols=7::20 43.5 50::20
text 2,12,"Sold To",cgtimes,italic,10,center,cols=5
cfont 8,11,40,11,cgtimes,bold,10,left
cfont 8,12,40,15,cgtimes,bold,10                          # sold to address
text 45,12,"Ship To",cgtimes,italic,10,center,cols=5
cfont 51,11,80,11,cgtimes,bold,10,left
text 51,12,{mcut(51,12,30,4,"","Y","Y")},cgtimes,bold,10

# ribbon section
cbox .5,18.5,80.5,22.5,5,crows=20.5::20,ccols=9 18 25 65
# special internal grid in ribbon box
cbox 29,18.5,65,21.5
cbox 42,18.5,56,21.5
text 1,19,"Order\nNumber",univers,italic,10,center,cols=8
text 10,19,"Order\nDate",univers,italic,10,center,cols=8
text 19,19,"Cust.\nNumber",univers,italic,10,center,cols=6
text 26,19,"Sls\nPrs",univers,italic,10,center,cols=3
text 30,19,"Purchase\nOrder No.",univers,italic,10,center,cols=12
text 43,19,"\nShip Via",univers,italic,10,center,cols=13
text 57,19,"Ship\nDate",univers,italic,10,center,cols=8
text 66,19,"\nTerms",univers,italic,10,center,cols=14

cfont 1,21,8,21,cgtimes,bold,10,center      # order #
cfont 10,21,17,21,cgtimes,bold,10,center    # order date
cfont 19,21,24,21,cgtimes,bold,10,center    # cust #
cfont 26,21,28,21,cgtimes,bold,10,left      # sls prs code
cfont 26,22,64,22,cgtimes,bold,10,left      # sls prs name
cfont 30,21,41,21,cgtimes,bold,10,center    # po #
cfont 43,21,55,21,cgtimes,bold,10,center    # ship via
cfont 57,21,64,21,cgtimes,bold,10,center    # ship date
cfont 66,21,80,22,cgtimes,10,center         # terms

# detail section
cbox .5,22.5,80.5,56.5,5,crows=24.5::10,ccols=5 10 16 51 55 67
text 1,23,"Qty\nOrd",univers,italic,10,right,cols=4
text 6,23,"Qty\nShip",univers,italic,10,right,cols=4
text 11,23,"Qty\nBkord",univers,10,italic,right,cols=4
text 17,23,"\nItem & Description",univers,italic,10
text 52,23,"\nU/M",univers,italic,10,center,cols=3
text 56,23,"Unit\nPrice",univers,italic,10,right,cols=11
text 68,23,"Extended\nPrice",univers,italic,10,right,cols=12

cfont 1,25,4,56,cgtimes,10,bold,right        # qty ord
cfont 6,25,9,56,cgtimes,10,bold,right        # qty shipped
cfont 11,25,15,56,cgtimes,10,bold,right      # qty b/o
cfont 17,25,50,56,cgtimes,10,left            # item # & desc
cfont 52,25,54,56,cgtimes,10,bold,center     # u/m
cfont 56,25,66,56,cgtimes,10,bold,right      # unit price
cfont 68,25,79,56,cgtimes,10,bold,right      # extended

# memo section
font "    @1,25,4,56",17,0,60,1,cgtimes,10,left
```

This text line adds a large text watermark on line 56, centered horizontally. The text is printed in cgtimes, 120 point, with 10% shading applied.

```
.
# watermark
text 1,56,"INVOICE",cgtimes,120,shade=10,center,cols=80,fit

# footer section
cbox 57,57,80.5,65,crows=59 63,ccols=67::20
text 58,58,"Sales Amt",univers,11
cfont 58,60,66,60,univers,11,left
text 58,61,"Sales Tax",univers,11
text 58,62,"Freight",univers,11
text 58,64.25,"TOTAL",univers,bold,14
cfont 68,58,79,65,cgtimes,bold,right,14                  # totals
```

The barcode command can be used to add barcodes in many symbologies. It is similar to other commands, in that you provide a column, row, and value. In addition, you specify a symbology (400 is Code 3 of 9), a point size or pixel height (14.0, being a decimal rather than integer value, is treated as point size), and a bar spacing value in pixels. Like most commands, you can use expressions in the value element of the command. In this example, the data stream data from column 9, row 11, for 6 characters is used on each page, using the get() function within an expression.

```
text 2,58,"Customer code as 3 of 9 barcode",univers,italic,10
barcode 2,58.67,{get(9,11,6)},400,14.0,4
```

The following lines produce different output for each of the two copies. Copy 1 is labeled with a text command to say it is the "Customer Copy", while copy 2 is labeled as "Accounting Copy". Any commands outside of 'if copy' blocks are applied to all copies.

```
# copy name section
if copy 1
    text 1,65.5,"Customer Copy",univers,12,bold,center,cols=80
end if
if copy 2
    text 1,65.5,"Accounting Copy",univers,12,bold,center,cols=80
end if
```

SIMPLE4 – invoice rule set (simple.rul)

This rule set demonstrates the use of constants, graphical shading, colors, and expressions to produce explanatory notes in the document. To produce this sample, use this command:

uf80c –i sample1.txt –f simple.rul –r simple4 –p pdf –o client:simple4.pdf

A title header prefixes all rule sets, which is just a unique name enclosed in brackets.

```
[simple4]
# to use this rule set, you need to FORCE the rule set with the -r option
# or remark (#) out the detect command in the rule sets above.
#
# This rule set takes the rule set above and improves it by adding
# constants, graphical shading, increases the resolution,
# and adds explanatory text commands for cust # and ship to #.
#
# Also adds a copy for a packing slip with no prices.

dsn_sample "sample1.txt"
detect 61,5,"~../../..  ......."        # invoice date and #
detect 9,11,"~......"                   # customer code
detect 10,21,"~../../.. ......"         # ord date and cust code
```

Constants are simple text names that are replaced by values later in the rule set. They can be used to simplify maintenance of the rule set, or to make it easier to read. In this example, a series of constants is defined using the const command, and you will find the names referenced throughout the balance of the rule set.

```
const MAXCOLS=80                        # max cols to output
const MAXRCOLS=79                       # MAXCOLS-1
const LEFTCOL=.5                        # use 1 if empty
const RIGHTCOL=80.5                     # use LEFTCOL for symmetry
const MAXROWS=66                        # max rows to output

cols MAXCOLS
rows MAXROWS

# copies
const CUSTOMER_COPY=1
const FILE_COPY=2
const PACK_COPY=3
pcopies 3
```

The dpi setting applies to laser output only, and instructs the printer to produce output at 600 dpi, providing a typically crisper, more professional look. In addition, the gs on *command turns on graphical shading mode, so that shade regions and shaded text are rendered as graphical data rather than using pcl's internal, typically coarse, shade macros.*

```
dpi 600
gs on                                   # turn on graphical shading

# enhancement constants
const HSHADE=30
const ISHADE=20
const DSHADE=10
const HFONT=univers,11
const IFONT=univers,italic,10
const DFONT=cgtimes,10
const DBFONT=DFONT,bold

# header section
cbox LEFTCOL,.5,RIGHTCOL,{MAXROWS+.5},5
image 1,1,12,6,"sdsilogo.pcl"
text 1,2,"Company Name",HFONT,14,bold,center,cols=MAXRCOLS
text 1,3,"Company Address\nCompany City, St Zipcode\nCompany
Phone",HFONT,12,bold,center,cols=MAXRCOLS
text 1,6,"Web: www.myweb.com\nEmail:
sales@myweb.com",HFONT,bold,center,cols=MAXRCOLS
text 1,2,"INVOICE",HFONT,16,bold,right,cols=MAXRCOLS

# invoice # section
cbox 67,4,RIGHTCOL,10,1,crows=6 8,ccols=74::ISHADE
text 68,5,"Date",IFONT
text 68,7,"Invoice",IFONT
text 68,9,"Page #",IFONT
# cut data from old position and place in new
text 75,5,{cut(61,5,8,"")},DBFONT
text 75,7,{cut(71,5,7,"")},DBFONT
text 75,9,{cut(79,5,2,"")},DBFONT
```

The cbox command shown here uses constants defined above, plus shows the use of color options, which are supported by PDF and color laser output. In this example, the interior is colored in cyan, and the lines are colored in blue. Alternately, RGB hex triplets (such as 800000 for dark red) can be specified using the rgb, scolor rgb, or lcolor rgb options.

```
# bill to / ship to section
cerase 1,11,MAXCOLS,11    # erase cust#,ship# used later in text commands
cbox LEFTCOL,11,RIGHTCOL,18.5,5,cyan,lcolor=blue,ccols=7::ISHADE 43.5 50::ISHADE
text 2,12,"Sold To",IFONT,center,cols=5
cfont 8,11,40,11,DBFONT,left
cfont 8,12,40,15,DBFONT                         # sold to address
```

This text command shows an example of how to use an expression to construct a message using a combination of hard-coded text and information from the data stream. In this example, the phrase "Your customer code is" is concatenated with the data at column 9, row 11, for 6 characters, on each page, and the result is printed at column 9, row 18, using the specifications provided by the constant IFONT, defined earlier in the rule set.

```
text 8,18,{"Your customer code is "+get(9,11,6)+"."},IFONT

text 45,12,"Ship To",IFONT,center,cols=5
cfont 51,11,80,11,DBFONT,left
text 51,12,{mcut(51,12,30,4,"","Y","Y")},DBFONT
text 51,18,{"Your ship to code is "+get(55,11,6)+"."},IFONT
```

```
# ribbon section
cbox LEFTCOL,18.5,RIGHTCOL,22.5,5,lcolor=blue,crows=20.5::ISHADE:cyan,ccols=9 18 25
65
# special internal grid in ribbon box
cbox 29,18.5,65,21.5
cbox 42,18.5,56,21.5
text 1,19,"Order\nNumber",IFONT,center,cols=8
text 10,19,"Order\nDate",IFONT,center,cols=8
text 19,19,"Cust.\nNumber",IFONT,center,cols=6
text 26,19,"Sls\nPrs",IFONT,center,cols=3
text 30,19,"Purchase\nOrder No.",IFONT,center,cols=12
text 43,19,"\nShip Via",IFONT,center,cols=13
text 57,19,"Ship\nDate",IFONT,center,cols=8
text 66,19,"\nTerms",IFONT,center,cols=14

cfont 1,21,8,21,DBFONT,center      # order #
cfont 10,21,17,21,DBFONT,center    # order date
cfont 19,21,24,21,DBFONT,center    # cust #
cfont 26,21,28,21,DBFONT,left      # sls prs code
cfont 26,22,64,22,DBFONT,left      # sls prs name
cfont 30,21,41,21,DBFONT,center    # po #
cfont 43,21,55,21,DBFONT,center    # ship via
cfont 57,21,64,21,DBFONT,center    # ship date
cfont 66,21,80,22,DBFONT,center    # terms

# detail section
if copy PACK_COPY
     erase "~\.[0-9][0-9]@62,25,79,56",-6,0,11,1
endif
cbox LEFTCOL,22.5,RIGHTCOL,56.5,5,crows=24.5::DSHADE,ccols=5 10 16 51 55 67
text 1,23,"Qty\nOrd",IFONT,right,cols=4
text 6,23,"Qty\nShip",IFONT,right,cols=4
text 11,23,"Qty\nBkord",IFONT,right,cols=4
text 17,23,"\nItem & Description",IFONT
text 52,23,"\nU/M",IFONT,center,cols=3
text 56,23,"Unit\nPrice",IFONT,right,cols=11
text 68,23,"Extended\nPrice",IFONT,right,cols=12

cfont 1,25,4,56,DBFONT,right        # qty ord
cfont 6,25,9,56,DBFONT,right        # qty shipped
cfont 11,25,15,56,DBFONT,right      # qty b/o
cfont 17,25,50,56,DFONT,left        # item # & desc
cfont 52,25,54,56,DBFONT,center     # u/m
cfont 56,25,66,56,DBFONT,right      # unit price
cfont 68,25,79,56,DBFONT,right      # extended

# memo section
font "    @1,25,4,56",17,0,60,1,DFONT,left

# watermark
if copy CUSTOMER_COPY,FILE_COPY
     text 1,56,"INVOICE",DFONT,120,shade=DSHADE,center,cols=MAXCOLS,fit
endif
if copy PACK_COPY
     text 1,56,"PACK SLIP",DFONT,120,shade=DSHADE,center,cols=MAXCOLS,fit
endif

# footer section
cbox 57,57,RIGHTCOL,65,lcolor=red,crows=59 63,ccols=67::HSHADE
text 58,58,"Sales Amt",HFONT
```

```
cfont 58,60,66,60,HFONT,left
text 58,61,"Sales Tax",HFONT
text 58,62,"Freight",HFONT
text 58,64.25,"TOTAL",HFONT,bold,14
cfont 68,58,79,65,DBFONT,right,14                  # totals

text 2,58,"Customer code as 3 of 9 barcode",IFONT
barcode 2,58.67,{get(9,11,6)},400,14.0,4
```

Note the use of constants to make this section easier to read.

```
# copy name section
if copy CUSTOMER_COPY
    text 1,65.5,"Customer Copy",HFONT,12,bold,center,cols=MAXCOLS
end if
if copy FILE_COPY
    text 1,65.5,"Accounting Copy",HFONT,12,bold,center,cols=MAXCOLS
end if
if copy PACK_COPY
    text 1,65.5,"Packing Slip",HFONT,12,bold,center,cols=MAXCOLS
end if
```

This text line demonstrates the use of multi-line text forced to fit within a certain number of columns. UnForm scans each of the two lines (delimited by the \n character sequence, or it could contain data with line-feed or carriage-return line-feed delimiters) to determine the width, beginning with the point size 12 specified in the command. The size is reduced until both lines will fit within the 20 columns specified with the cols option. Once the correct point size is determined, the lines are spaced normally for that height. For example, if the size required is 8.25 points, then the lines will be spaced 8.25 points apart. If spacing had been set to 1.5, then the lines would be spaced 12.33 points apart.

```
text 2,62,"This sample message text, which contains\nline breaks,  is sized to fit
in 20 columns.",cols 20,cgtimes,12,fit,spacing 1
```

INVOICE - invoice for pre-printed form (advanced.rul)

This sample is an invoice that is intended for a pre-printed form. The data generated by the application doesn't include any headings or simulated line drawing like a plain-paper invoice might. In this case, UnForm must simulate the entire pre-printed invoice form.

uf80c –i sample1.txt –f advanced.rul –p pdf -o client:invoice.pdf

A title header prefixes all rule sets, which is just a unique name enclosed in brackets.

```
[Invoice]
```

Detect statements are used to distinguish this form from any other report that the application might send to the printer through UnForm. Unlike most form packages, UnForm doesn't dedicate a printer name to a particular form (though it can be configured to do so). Instead, it reads the first page of data, then compares it to the detect statements found in the various rule sets in the rule file.

The detect statements below indicate that

- *a date (mm/dd/yy format) followed by 2 spaces, followed by 7 more characters will appear at column 61, row 5*
- *6 characters will appear at column 9, row 11*
- *a date, a space, and 6 characters will appear at column 10, row 21*

```
detect 61,5,"~../../..  ......."        # invoice date and #
detect 9,11,"~......"                   # customer code
detect 10,21,"~../../.. ......"         # ord date and cust code
```

The following lines define several constants that are used elsewhere in the rule set. Wherever the constant names appear in a command, the value is substituted. Constants are not variables and are not interpreted while the job is processed. They are simply literal placeholders used while UnForm reads rule set lines.

```
# set up document constants
const MAXCOLS=80                        # max cols to output
const MAXRCOLS=79                       # MAXCOLS-1
const LEFTCOL=.5                        # use 1 if empty
const RIGHTCOL=80.5                     # use LEFTCOL for symmetry
const MAXROWS=66                        # max rows to output
```

The following lines define the page size and orientation. The dpi command sets the printer to 600 dots per inch. The rows and cols commands set the dimension for positioning and scaling. All positioning will be based on 80 columns and 66 rows appearing within the printed margins of the page. The gs *on* *command triggers the use of graphical shading, which improves the look of shade regions over the native pcl shading of most laser printers, especially at higher dpi settings and shade percentages. In addition, UnForm will generate two copies of the job, with each page producing two copies as processed (collated).*

```
portrait
dpi 600
gs on                                   # graphical shading
cols MAXCOLS                            # max output columns
rows MAXROWS                            # max output rows

# to print more copies, increase value and add copy titles in prejob
pcopies 2                               # max # of copies
```

If this rule set is used to produce a PDF document, then the title of "Sample Invoice" will be added to the PDF file. For laser output, the title command is ignored.

```
title "Invoice Sample"                  # view in PDF properties
```

The prejob code block is executed once at the beginning of the job, after the first page of data has been read and the rule set parsed. This example is simply setting a variable form_title$ *to a literal value INVOICE. This variable is used later in the rule set.*

The prepage code block is executed once per page, just after UnForm has read the text for the page, but before any copies of that page have been printed. Within a prepage code block, you can insert any valid Business Basic code (though you need to be careful not to insert any UnForm commands.) This code initializes a variable shipzip$ to null, then looks for a regular expression pattern of 5 digits on line 15. If the pattern is found, it sets shipzip$ to the zip code. After the code block is closed, a barcode command is used to place a postnet barcode below the shipping address. The barcode command uses the syntax "{shipzip$}", indicating the expression shipzip$ should be used to generate the data to barcode.

Once the prepage code block creates shipzip$, it then scans a range of rows looking for special memo format lines. It marks these lines with the characters "mL" in the first two columns. Later in the rule set, you'll see how these markers are used to treat memo lines differently than standard invoice lines.

The order of execution is controlled by UnForm. There is actually no need to place the barcode command below the prepage code block, as UnForm will properly execute the code block before any form commands are executed at run-time.

```
prejob {
     # set up variables needed by merged routines below
     # if form title changes per page,
     # set up in prepage routine below
     form_title$="INVOICE"
}

prepage {
     # find zip code in city,state,zip line for bar code
     shipzip$=""
     # regular expression of 5 digits on line 15
     x=mask(text$[15],"[0-9][0-9][0-9][0-9][0-9]")
     if x>0 then shipzip$=get(x,15,5)

     # mark memo lines for special handling in detail section below
     # memo start in column 28 with all spaces before
     for i=25 to 56
          if len(text$[i])>27 and trim(text$[i](1,27))="" then \
               text$[i](1,2)="mL"
     next i
}
```

The pdf driver supports the ability to email the PDF file created using the email command. The commented # email line below provides an example of the command. It requires four arguments, each of which can be a literal string value or a string expression enclosed in braces. In order for the email command to work, the mailcall.ini file must be properly configured for your system.

```
# When run in PDF mode, and if mailcall.ini is configured properly,
# and if the system can communicate with the mail server, then the
# next line would send the PDF invoice as an attachment to an email.
# email "someone@somewhere.com","me@mycompany.com", \
#    {"A test invoice "+cvs(get(71,5,7),3)}, \
#    "Attached is a sample invoice\n"
```

The next group of commands creates a page header with box and text commands. The box commands are given as the cbox variant, which accepts two pairs of numbers as opposite corners of the box. Some of the commands are stored in a different rule set, called "Mrg Form Header". This rule set is also located in the advanced.rul file. The lines in that rule set are merged in here as if they were part of this rule set.

Note that some of the text commands, and also a barcode command, use an expression rather than a literal. An expression is an executable value assignment enclosed in braces. For example, one text command uses an expression {cut(61,5,8,"")}*,which cuts out the text at column 61, row 5, for 8 columns, returning the result, while setting those positions to "". The result is printing at position 75,5 what was at position 61,5.*

```
# heading section
const HFONT=univers,12                        # headings
cbox LEFTCOL,1,RIGHTCOL,MAXROWS,5             # complete page box
merge "Mrg Form Header"                       # merge std hdr rules
```

```
# right top ribbon
const HFONT=univers,11,italic                # headings
const DFONT=cgtimes,11,bold                  # data

# draw info box with internal grid and shading
# horizontal lines at 6 and 8
# vertical line at 74 with shading between 67 and 74
cbox 67,4,RIGHTCOL,10,5,crows=6 8,ccols=74::20
text 68,5,"Date",HFONT
text 68,7,"Invoice",HFONT
text 68,9,"Page #",HFONT
# cut data from old position and place in new
text 75,5,{cut(61,5,8,"")},DFONT
text 75,7,{cut(71,5,7,"")},DFONT
text 75,9,{cut(79,5,2,"")},DFONT

# sold to section
cbox LEFTCOL,10,41,18.5,5
cbox LEFTCOL,10,41,11.25,0,10
text 8,10.75,"SOLD TO",HFONT,bold
cfont 8,12,40,15,DFONT                       # sold to address
if copy 1
     barcode 8,16,{shipzip$},900,9.0,2
end if
text 2,18,{"Your customer code is "+cut(9,11,6,"")+"."},8,cgtimes

# ship to section
cbox 41,10,RIGHTCOL,18.5,5
cbox 41,10,RIGHTCOL,11.25,0,10
text 48,10.75,"SHIP TO",HFONT,bold
# cut ship to address and place in new position
text 48,12,{mcut(51,12,30,4,"","Y","Y")},DFONT
text 43,18,{"Your ship to code is "+cut(55,11,6,"")+"."},8,cgtimes
```

This section draws order detail boxes and headings. The first cbox command draws a grid, using the internal crows and ccols options. In addition to the boxes and headings, the font used for the data from the input stream is changed using a series of cfont commands, one for each section.

```
# ribbon section
const L1=19
const L2=20
# draw info box with internal grid and shading
# horizontal line at 20.5 with shading between 18.5 and 20.5
# vertical lines at 9, 18, 25, and 65
cbox LEFTCOL,18.5,RIGHTCOL,22.5,5,crows=20.5::20,ccols=9 18 25 65
# special internal grid in ribbon box
cbox 29,18.5,65,21.5
cbox 42,18.5,56,21.5
# ribbon headings
text 1,L1,"Order",HFONT,right,cols=8
text 1,L2,"Number",HFONT,right,cols=8
text 10,L1,"Order",HFONT,center,cols=8
text 10,L2,"Date",HFONT,center,cols=8
text 19,L1,"Cust.",HFONT
text 19,L2,"Number",HFONT
text 26,L1,"Sls",HFONT
text 26,L2,"Prs",HFONT
text 30,L1,"Purchase",HFONT
text 30,L2,"Order No.",HFONT
text 43,L2,"Ship Via",HFONT
text 57,L1,"Ship",HFONT,center,cols=8
text 57,L2,"Date",HFONT,center,cols=8
text 66,L2,"Terms",HFONT
# ribbon data
cfont 1,21,8,21,DFONT,right                         # order #
cfont 10,21,17,21,DFONT,center                      # order date
cfont 19,21,24,21,DFONT                             # cust #
cfont 26,21,28,21,DFONT                             # sls prs code
cfont 26,22,64,22,DFONT                             # sls prs name
cfont 30,21,41,21,DFONT                             # po #
cfont 43,21,55,21,DFONT                             # ship via
cfont 57,21,64,21,DFONT,center                      # ship date
cfont 66,21,MAXCOLS,22,DFONT                        # terms
```

This section of lines controls the formatting of the invoice detail lines. A grid is drawn around the column headers and detail lines. The column headers are shaded. Item detail lines are fonted using a series of font commands that look for the pattern "~\.[0-9][0-9][0-9][0-9]" which is a period followed by 4 digits. Wherever that occurs, font changes are made relative to that position. Similarly, the memo lines identified by the prepage code block and marked with the text marker "mL" are fonted with a different column structure. In addition to the font command, an erase command is used to remove the text markers.

```
# detail section
# detail headings
const L1=23
const L2=24
# draw info box with internal grid and shading
# horizontal line at 24.5 with shading between 22.5 and 24.5
# vertical lines at 5, 10, 16, 51, 55, and 67
cbox LEFTCOL,22.5,RIGHTCOL,56.5,5,crows=24.5::10, \
     ccols=5 10 16 51 55 67
text 1,L1,"Qty",HFONT,right,cols=4
text 1,L2,"Ord",HFONT,right,cols=4
text 6,L1,"Qty",HFONT,right,cols=4
text 6,L2,"Ship",HFONT,right,cols=4
text 11,L1,"Qty",HFONT,right,cols=5
text 11,L2,"Bkord",HFONT,right,cols=5
text 20,L2,"Item & Description",HFONT
text 52,L2,"U/M",HFONT,center,cols=3
text 56,L1,"Unit",HFONT,right,cols=11
text 56,L2,"Price",HFONT,right,cols=11
text 68,L1,"Extended",HFONT,right,cols=12
text 68,L2,"Price",HFONT,right,cols=12
# detail data
# Modify fonts for lines. As comments may be present in the same rows,
# use a pattern to locate the .nnnn in the price column,
# which indicates a part number line.
# Use a prepage routine to find the comments and change their font.
font "~\.[0-9][0-9][0-9][0-9]",-61,0,4,1,DFONT,right   # qty ord
font "~\.[0-9][0-9][0-9][0-9]",-56,0,4,1,DFONT,right   # qty shipped
font "~\.[0-9][0-9][0-9][0-9]",-50,0,4,1,DFONT,right   # qty b/o
font "~\.[0-9][0-9][0-9][0-9]",-42,0,30,2,DFONT        # item # & desc
font "~\.[0-9][0-9][0-9][0-9]",-10,0,3,1,DFONT,center  # u/m
font "~\.[0-9][0-9][0-9][0-9]",-6,0,11,1,DFONT,right   # unit price
font "~\.[0-9][0-9][0-9][0-9]",6,0,12,1,DFONT,right    # ext price

# handle memo lines
# inserted 'mL' in prepage above
font "mL@1,25,2,56",10,0,63,1,HFONT
erase "mL@1,25,2,56",0,0,2,1
```

Watermark text is placed in the middle of the detail lines. This text is centered between column 1 and MAXCOLS, is rendered at 120 points, and is printed at 20% gray shading.

```
# watermark - large font with light shading
text 1,52,{form_title$},cgtimes,120,shade 20,center,cols=MAXCOLS
```

The totals section is formatted like the other sections, with a grid, text headings, and font changes that apply to the input stream text.

```
# totals section
# draw info box with internal grid and shading
# horizontal lines at 59 and 63
# vertical line at 69 with shading between 58 and 69
cbox 58,57,RIGHTCOL,65,5,ccols=69::20,crows=59 63
text 59,58,"Sales Amt",HFONT
text 59,61,"Sales Tax",HFONT
text 59,62,"Freight",HFONT
text 59,64.25,"TOTAL",HFONT,bold,14
cfont 59,60,68,60,HFONT                                # disc hdr
cfont 70,58,MAXRCOLS,65,DFONT,14,decimal               # totals
```

These text lines simply demonstrate some of UnForm's paragraph features. The first text command forces the longest line in the paragraph to fit within the number of defined columns. The maximum point *size is 12, but that may be adjusted down to accommodate the longest line. Lines are delimited by the \n character sequence, or by a CHR(10) within an expression. Line spacing is determined by the final point size, and may be adjusted with the spacing option. For example, if the rendered size is 8 point, then the spacing of 1 will result in 9 lines per inch (9 x 8=72), while spacing of 1.5 would result in 6 lines per inch (9/1.5=6).*

The second example will force use the defined point size to render the text, but any lines wider than the specified columns will be word-wrapped.

The third example shows how to use a specified ASCII value in a text command. The ASCII value 174, when printed using the symbol set 9J, is a trademark symbol. This technique can be used to print Latin characters and special symbols. The symbol set determines what any given character value prints as. The 9J symbol set is the default. See the –testpr command line option for viewing printed tables of different symbol sets.

```
# footer section
# These lines show fitting and wrapping of text
text 2,60,"This sample message text, which contains\nline breaks, \
     is sized to fit in 20 columns.",cols 20,cgtimes,12, \
     fit,spacing 1

text 28,60,"This sample message text is word wrapped to not exceed \
     20 columns, while retaining the specified 12 point size.",\
     cgtimes,cols 20,12,wrap,spacing 1

text 2,64,"This sample was generated by UnForm<174>.",7,cgtimes, \
          symset 9J,blue
```

This set of commands places the phrase "Customer Copy" on copy 1, and "Remittance Copy" on copy 2. The text is placed at row 65.5, and is centered within the columns defined at column 1 and the constant MAXCOLS, which represents the whole page width.

```
# copy name section
const ROW=65.5
if copy 1
     text 1,ROW,"Customer Copy",HFONT,bold,center,cols=MAXCOLS
end if
if copy 2
     text 1,ROW,"Accounting Copy",HFONT,bold,center,cols=MAXCOLS
end if
```

STATEMENT - plain paper form, two page formats in same job (advanced.rul)

In this sample, a two-page, plain paper statement is printed. The two pages contain two slightly different formats, with the second page containing detail lines and a customer aging, and the first page containing some more detail lines and the phrase "CONTINUED" at the bottom. In the same statement print run, some statements may contain a single page, others two or more pages.

The trick here is to get UnForm to produce two formats based on the content of each page. In order to accomplish this, we define the job to produce multiple copies, and assign certain copies to certain formats. Using a precopy{} code block, we can then control the printing of the different formats.

uf80c –i sample2.txt –f advanced.rul –p pdf –o client:statement.pdf

This statement header identifies this rule set.

```
[Statement]
```

The word STATEMENT appears at column 34, row 2, and a date appears at column 65, row 7. To further clarify, a date format is matched at position 65, 7.

```
detect 34,2,"STATEMENT"
detect 65,7,"~../../.."                       # statement date
```

The page dimensions are 66 rows and 75 columns. The text input to UnForm doesn't contain any form-feeds to indicate the end of a page, so the command "page 66" tells UnForm to consider each 66 lines to be a page.

Pcopies 4 is used to tell UnForm to print 4 copies of each page, with copies following each other in sequence for each page (collated). You will find later that UnForm doesn't actually print all copies of each page, but instead simply prints selected copies, depending on the format required. As each page is processed, if the page contains aging totals, UnForm prints 2 copies of that format, and if it does not contain aging totals, then UnForm prints 2 copies of the second format.

```
# set up document constants
const MAXCOLS=75                              # max cols to output
const MAXRCOLS=74                             # MAXCOLS-1
const LEFTCOL=1                               # use 1 if empty
const RIGHTCOL=75                             # MAXCOLS for symmetry
const MAXROWS=66                              # max rows to output

portrait
dpi 300
```

```
gs on                                   # graphical shading
cols MAXCOLS                            # max output columns
rows MAXROWS                            # max output rows
page MAXROWS                            # no form-feeds used

# to print more copies, increase value and add copy titles in prejob
# Copy 1,2     Statement with aging totals
# Copy 3,4     Statement w/o  aging totals
pcopies 4                               # max # of copies
```

If this rule set is used to produce a PDF document, then the title "Statement Sample" will be added to the PDF file. For laser output, the title command is ignored.

```
title "Statement Sample"                # view in PDF properties
```

The prejob command defines a string variable form_title$, assigning it the value "STATEMENT". This variable is used later in the rule set for a page heading and also in a watermark.

```
prejob {
     # set up variables needed by merged routines below
     # if form title changes per page,
     # set up in prepage routine below
     form_title$="STATEMENT"
}
```

The prepage code block performs 2 functions. It checks the input data for the word "CONTINUED" at position 66, 64. If that word is present, then a variable continued$ is assigned to the phrase "Continued"; otherwise it is set to null. In addition, at three individual lines (16, 62, and 64), there may be single ! characters used as character-mode vertical lines in the input data. Elsewhere in the rule set is a 'vline "!!", erase' command, which erases instances of 2 or more ! characters vertically on the page. This code takes care of the single-row instances.

```
prepage {
     # get continued if it exists
     continued$=get(66,64,9)
     if continued$="CONTINUED" then continued$="Continued" \
          else continued$=""

# remove single ! from line draw regions
        x=pos("!"=text$[16]); \
        while x>0; text$[16](x,1)="",x=pos("!"=text$[16]);wend

        x=pos("!"=text$[62]); \
        while x>0; text$[62](x,1)="",x=pos("!"=text$[62]);wend
        x=pos("!"=text$[64]); \
        while x>0; text$[64](x,1)="",x=pos("!"=text$[64]);wend

}
```

The precopy code block is executed as each of the 4 copies are about to be printed. The logic here indicates the copies 1 and 2 are for pages that do not contain the word "CONTINUED" (remember the prepage code block?), and copies 3 and 4 do contain that word. By setting the variable skip to a non-zero value, the copy being processed is skipped. Only 1 of the 2 formats is printed, depending on the content of the page.

```
precopy {
     if copy=1 or copy=2 then if continued$="Continued" then skip=1
     if copy=3 or copy=4 then if continued$<>"Continued" then skip=1
}
```

The following lines remove most of the existing character-mode line drawing elements from the input data. The hline and vline commands scan for places where at least the indicated number of characters, horizontally or vertically, occur on the page. The erase option removes them rather than replacing them with graphical lines.

```
#remove existing lines
hline "--",erase
hline "==",erase
vline "!!",erase
cerase 1,1,1,MAXROWS                         # erase all 1st column
cerase MAXCOLS,1,MAXCOLS,MAXROWS             # erase all last column
```

The following lines draw the page headings. Some of the commands are stored in another rule set, "Mrg Form Header", which is merged as the rule set is parsed. The headings already exist, and are moved and fonted with text commands using expressions, such as {cut(66,5,4,"")}.

```
# heading section
const HFONT=univers,12                       # headings
cerase 1,1,MAXCOLS,10
cbox LEFTCOL,1,RIGHTCOL,MAXROWS,5            # complete page box
merge "Mrg Form Header"                      # merge std hdr rules

# right top ribbon section
const HFONT=univers,11,italic                # headings
const DFONT=cgtimes,11,bold                  # data
# draw info box with internal grid and shading
# horizontal line at 6
# vertical line at 68 with shading between 63 and 68
cbox 63,5,MAXCOLS,9,5,crows=7,ccols=68::20
```

```
text 64,6,{cut(66,5,4,"")},HFONT             # page #
text 64,8,{cut(59,7,4,"")},HFONT             # date
text 69,6,{trim(cut(71,5,3,""))},DFONT       # page #
text 69,8,{trim(cut(65,7,8,""))},DFONT       # date

# customer section
# draw info box with internal grid and shading
# vertical line at 10 with shading between 1 and 10
cbox LEFTCOL,10,MAXCOLS,15,5,ccols=10::10
text 2,11,{cut(2,10,2,"")},HFONT             # to
text 4,13,{trim(cut(15,10,10,""))},DFONT          # cust code
cfont 12,11,MAXCOLS,14,DFONT                 # address
```

The detail section contains several columns of information. There are fewer detail lines on pages with the aging data, so the grid drawing is made specific to particular formats with "if copy 1,2" and "if copy 3,4" sections. Then two groups of font changes are used, first for the column headings and then for the data columns.

```
# detail section
# detail headings
# draw info box with internal grid and shading
# horizontal line at 6
# vertical line at 68 with shading between 63 and 68
if copy 1,2
     cbox LEFTCOL,15,MAXCOLS,56,5,crows=17::20, \
          ccols=10 18 27 39 48 60 63
end if
if copy 3,4
     cbox LEFTCOL,15,MAXCOLS,61,5,crows=17::20, \
          ccols=10 18 27 39 48 60 63
end if
const ROW=16
cfont 2,ROW,9,ROW,HFONT,center                    # date
cfont 11,ROW,17,ROW,HFONT                         # inv #
cfont 19,ROW,26,ROW,HFONT,center                  # due date
cfont 28,ROW,38,ROW,HFONT,right                   # due amt
cfont 40,ROW,47,ROW,HFONT,center                  # pmt date
cfont 49,ROW,59,ROW,HFONT,right                   # pmt amt
cfont 61,ROW,62,ROW,HFONT,center                  # type
cfont 64,ROW,MAXRCOLS,ROW,HFONT,right             # balance
# detail data
const DFONT=cgtimes,11                            # data
cfont 2,18,9,60,DFONT,center                      # date
cfont 11,18,17,60,DFONT                           # inv #
cfont 19,18,26,60,DFONT,center                    # due date
cfont 28,18,38,60,DFONT,right                     # due amt
cfont 40,18,47,60,DFONT,center                    # pmt date
cfont 49,18,59,60,DFONT,right                     # pmt amt
cfont 61,18,62,60,DFONT,center                    # type
cfont 64,18,MAXRCOLS,60,DFONT,right,BOLD          # balance
```

A watermark prints the form title as large, lightly shaded text. Its position depends upon the format, hence the use of if copy blocks.

```
# watermark - large font with light shading and rotation
if copy 1,2
     text 39,56,{form_title$},cgtimes,75,shade 20,center, \
          cols=MAXCOLS,rotate 90
end if
if copy 3,4
     text 44,61,{form_title$},cgtimes,85,shade 20,center, \
          cols=MAXCOLS,rotate 90
end if
```

The footer section differs considerably between the two formats. Copies 1 and 2 are associated with pages that have aging data, so you see the fonting of the aging columns defined there. Copies 3 and 4 are printed when the word "CONTINUED" appears, and that word is printed below, though as the value stored in continued$ ("Continued").

```
# footer section
# remarks
if copy 1,2
     cbox LEFTCOL,56,RIGHTCOL,61,5
     cfont 2,57,MAXRCOLS,60,HFONT
endif
# totals
const DFONT=cgtimes,11,bold                       # data
if copy 1,2
     cbox LEFTCOL,61,RIGHTCOL,64.5,5,crows=63::20, \
          CCOLS=14 26 38 50 62
     const ROW=62
     cfont 1,ROW,13,ROW,HFONT,right                    # current
     cfont 15,ROW,25,ROW,HFONT,right                   # 1-15
     cfont 27,ROW,37,ROW,HFONT,right                   # 16-30
     cfont 39,ROW,49,ROW,HFONT,right                   # 31-45
     cfont 51,ROW,61,ROW,HFONT,right                   # over 45
     cfont 63,ROW,MAXRCOLS,ROW,HFONT,right,bold,12     # total due
     const ROW=64
     cfont 1,ROW,13,ROW,DFONT,right                    # current
     cfont 15,ROW,25,ROW,DFONT,right                   # 1-15
     cfont 27,ROW,37,ROW,DFONT,right                   # 16-30
     cfont 39,ROW,49,ROW,DFONT,right                   # 31-45
     cfont 51,ROW,61,ROW,DFONT,right                   # over 45
     cfont 63,ROW,MAXRCOLS,ROW,DFONT,right,bold,12     # total due
endif
```

```
if copy 3,4
     cerase 1,62,MAXCOLS,66
     text 1,65,{Continued$},HFONT,right,cols=MAXRCOLS
endif
```

Finally, within the two formats are two physical copies. Each of these copies is either for the customer to keep or for the customer to return with their payment. Copy 1, the first page of format 1, and copy 3, the first page of format 2, get the "Customer Copy" footer. The others get the "Remittance Copy" footer.

```
# copy name section
const ROW=65.5
if copy 1,3
     text 1,ROW,"Customer Copy",HFONT,bold,center,cols=MAXCOLS
end if
if copy 2,4
     text 1,ROW,"Remittance Copy",HFONT,bold,center,cols=MAXCOLS
end if
```

aging report - enhanced Aging report (advanced.rul)

In this third example, an aging report is enhanced to be more readable. This shows the use of *relative* enhancements, which are those applied relative to the occurrence of text or regular expressions anywhere on the page.

uf80c –i sample3.txt –f advanced.rul –p pdf –o client:aging.pdf

This statement header identifies this rule set.

```
[AgingReport]
```

The only detect statement required is this one, looking for the report title at column 50, row 2.

```
detect 50,2,"Detail Aging Report"
```

These constants are used throughout the rule set.

```
# set up document constants
const MAXCOLS=131                       # max cols to output
const MAXRCOLS=130                      # MAXCOLS-1
const LEFTCOL=.5                        # use 1 if empty
const RIGHTCOL=131.5                    # LEFTCOL for symmetry
const MAXROWS=66                        # max rows to output
```

This report should print in landscape orientation, rather than the default portrait. UnForm will scale the columns and rows to 131 by 66.

```
landscape
dpi 1200
gs on                                   # graphical shading
cols MAXCOLS                            # max output cols
rows MAXROWS                            # max output rows

pcopies 1                               # max # of copies
```

The title "Aging Sample" will appear in PDF document properties. It is ignored for laser output.

```
title "Aging Sample"                    # view in PDF properties
```

The following prejob code demonstrates the use of sdOffice™ to mine data from this report and export it to Microsoft Excel®. SdOffice can be running anywhere on your network on a system with Excel. The code relies on your setting two variables correctly. First, the sdo$ variable should be set to the path to the sdOffice client program sdofc_e.bb. In addition, the value of gbl("$sdhost") needs to be set to the address or hostname of the system running sdOffice. An optional way of doing this is to define an environment variable prior to running UnForm, called SDHOST. If you use that alternative, then comment out the x$=gbl("$sdhost") line.

The code here contains enough error handling to ignore the code if sdOffice isn't present or fails to execute.

```
prejob {
     # set up sdOffice export to Excel
     # set to path to your sdoffice *.pv programs
     sdo$="/u0/sdofc/sdofc_e.pv"

     # You can set the environment variable SDHOST, or use this
     # stbl function to define the sdOffice server address
     x$=gbl("$sdhost","bcj")

     # initialize excel
     call sdo$,err=prejob_done,"newbook","",errmsg$
     if errmsg$>"" then goto prejob_done
     sdofc_init=1
     call sdo$,"show","",""
     call sdo$,"setdelim |","",""
     call sdo$,"writerow ID|Name|Phone|Over 60|Total","",""
     call sdo$,"format row=1,font=Arial,size=12,bold","",""
prejob_done:
}
```

The prepage code block starts with code that exports data to Excel, but only if the prejob code block successfully initializes the sdOffice connection. In addition to that code, it also sets two numeric variables, colw and scol, based upon positions and widths of report column headers. These values are used later in the rule set for fonting and line drawing.

```
prepage{
     # if prejob hasn't initialized sdoffice, skip this code
     if sdofc_init<>1 then goto sdofc_complete

     for row=1 to 66
          ln$=text$[row]

          # customer heading row contain phone numbers
          x=mask(ln$,"\(...-...-....\)")
          while x
               custid$=mid(ln$,1,6)
               custname$=trim(mid(ln$,8,30))
               custphone$=trim(mid(ln$,38,14))
               x=0
          wend

          # totals - 50 plus spaces followed by digit-.-digit-digit
          x=mask(ln$,"^"+fill(50)+".*[0-9]\.[0-9][0-9]")
          while x
               amount60=cnum(mid(ln$,87,11))
               amount90=cnum(mid(ln$,98,11))
               amount120=cnum(mid(ln$,109,11))
               over60=amount60+amount90+amount120
               total=cnum(mid(ln$,120,11))

               export$=custid$+"|"+custname$+"|"+custphone$+"|"
               export$=export$+str(over60)+"|"+str(total)
               call sdo$,"writerow "+export$,"",""
               x=0
          wend

     next row
sdofc_complete:

     # Now for some tricky code.
     # Agings can have different headings and column widths
     # To use version 5 features allowing variable columns and rows,
     # the following code will calculate starting positions
     # and column widths. It assumes a consistency in column widths,
     # 1 char negative, and 1 blank space between each column
     hd1$=text$[7]                # temp heading line with agings
     x=pos("Type"=hd1$)
     xhd1$=trim(hd1$(x+4))        # remove all except agings
     x=pos(" "=xhd1$)
     x$=xhd1$(1,x-1)              # get first column header
     xhd1$=trim(xhd1$(x))
     x=pos(x$=hd1$)               # find true position
     x1=x+len(x$)-1               # get end of first column
     # now find end of 2nd column
     x=pos(" "=xhd1$)
     x$=xhd1$(1,x-1)              # get second column header
     x=pos(x$=hd1$)
     x2=x+len(x$)-1               # get end of second column
     # now calculate mask width less space between columns
     colw=x2-x1-1
     # now calculate start of first field
     scol=x1-colw+2
}
```

The postjob code block performs some closing formatting control if the job is exporting data to Excel. If sdOffice is not being used, based upon the attempt to initialize it in the prejob code block, then this code is skipped.

```
postjob{
     # if prejob hasn't initialized sdoffice, skip this code
     if sdofc_init<>1 then goto sdofc_complete2

     call sdo$,"leaveopen","",""
     call sdo$,"format autofit","",""
     call sdo$,"format col=1,numberformat=@","",""
     call sdo$,"format col=4,numberformat=""###,##0.00""","",""
     call sdo$,"format col=5,numberformat=""###,##0.00"",bold","",""

     call sdo$,"insertrow 1","",""
     call sdo$,"mergecells range=A1:E1","",""
     call sdo$,"writecell range=A1,value="+$22$+ \
          "Over 60 Aging Values as of "+date(0)+$22$,"",""
     call sdo$,"format range=A1:E1,center,size=15,bold","",""
sdofc_complete2:
}
```

Here, finally, are the commands to enhance the formatting of the report. The initial commands use text commands with cut expressions to move the report header data around and change the fonting.

```
# heading section
const BLFONT=univers,10,bold,italic
const BSFONT=univers,9,bold,italic
cbox .5,.5,RIGHTCOL,5,5,30
# line 1
text 2,1.25,{trim(cut(1,1,10,""))},BSFONT                   # date
text 1,1.25,{trim(cut(20,1,100,""))},BLFONT,center, \
     cols=MAXRCOLS  # comp name
text 1,1.25,{trim(cut(121,1,15,""))},BSFONT,right, \
     cols=MAXRCOLS  # page #
# line 2
text 2,2.35,{trim(cut(1,2,10,""))},BSFONT                   # time
text 1,2.35,{trim(cut(20,2,100,""))},BLFONT,center, \
     cols=MAXRCOLS  # rpt title
# line 3
text 1,3.45,{trim(cut(20,3,100,""))},BSFONT,center, \
     cols=MAXRCOLS  # sub heading
# line 4
text 1,4.45,{trim(cut(20,4,100,""))},BSFONT,center, \
     cols=MAXRCOLS  # sub heading
```

This section formats the column headings. The left portion is drawn with text commands, while the aging columns are fonted with font commands, which use the positions from the values calculated in the prepage code block.

```
# detail heading section
const HFONT=univers,10,italic
cbox LEFTCOL,5.25,RIGHTCOL,7.5,1,20
# line 1
cerase 1,6,MAXCOLS,6
text 1,6,"Customer # & Name",HFONT
text 38,6,"Phone #",HFONT,center,cols=14
text 54,6,"Contact",HFONT

# line 2
cerase 1,7,49,7
text 3,7,"Invoice #",HFONT
text 12,7,"Due Date",HFONT,center,cols=8
text 21,7,"P/O #",HFONT
text 32,7,"Order #",HFONT
text 39,7,"Terms",HFONT,center,cols=5
text 45,7,"Type",HFONT,center,cols=4
# using variables from prepage, enhance aging headings
font {scol},7,{colw-1},1,HFONT,right
font {scol+1*(colw+1)},7,{colw-1},1,HFONT,right
font {scol+2*(colw+1)},7,{colw-1},1,HFONT,right
font {scol+3*(colw+1)},7,{colw-1},1,HFONT,right
font {scol+4*(colw+1)},7,{colw-1},1,HFONT,right
font {scol+5*(colw+1)},7,{colw-1},1,HFONT,right
font {scol+6*(colw+1)},7,{colw},1,HFONT,right,bold
```

The report body is enhanced using UnForm's ability to scan for patterns and anchor enhancements to those patterns. The first series of font commands scan for two spaces in the region from column 1, row 9 through column 2, row 66 (defined as the constant MAXROWS above). At each point in that search region, if the two spaces are found, a font command is issued relative to the location. This changes the font of the input data at that location.

The second series of font commands looks for customer heading line types, by searching for any alpha or digit character in the region 1,9 though 2,66. A different set of font commands is then issued for those positions.

```
# detail data section
const BDFONT=cgtimes,10,bold
const DFONT=cgtimes,10
# invoice line
font "  @1,9,2,MAXROWS",2,0,8,1,DFONT
font "  @1,9,2,MAXROWS",11,0,8,1,DFONT,center
font "  @1,9,2,MAXROWS",20,0,10,1,DFONT
font "  @1,9,2,MAXROWS",31,0,7,1,DFONT
font "  @1,9,2,MAXROWS",38,0,5,1,DFONT,center
font "  @1,9,2,MAXROWS",44,0,4,1,DFONT,center
# using variables from prepage, enhance agings
font "  @1,9,2,MAXROWS",{scol},0,{colw},1,DFONT,decimal
font "  @1,9,2,MAXROWS",{scol+1*(colw+1)},0,{colw},1,DFONT,decimal
font "  @1,9,2,MAXROWS",{scol+2*(colw+1)},0,{colw},1,DFONT,decimal
font "  @1,9,2,MAXROWS",{scol+3*(colw+1)},0,{colw},1,DFONT,decimal
font "  @1,9,2,MAXROWS",{scol+4*(colw+1)},0,{colw},1,DFONT,decimal
font "  @1,9,2,MAXROWS",{scol+5*(colw+1)},0,{colw},1,DFONT,decimal
font "  @1,9,2,MAXROWS",{scol+6*(colw+1)},0,{colw+1},1,BDFONT,decimal

# cust line
font "~[A-Z0-9]@1,9,2,MAXROWS",0,0,6,1,BDFONT
font "~[A-Z0-9]@1,9,2,MAXROWS",7,0,28,1,BDFONT
font "~[A-Z0-9]@1,9,2,MAXROWS",37,0,14,1,BDFONT,center
font "~[A-Z0-9]@1,9,2,MAXROWS",53,0,36,1,BDFONT
shade "~[A-Z0-9]@1,9,2,MAXROWS",0,-.15,{RIGHTCOL-1.5},1,20
```

The following commands look for sequences of dashes, which indicate sub total lines. Wherever a sequence of 50 dashes occurs, a box is drawn and input data is bolded. In addition, the original dashes are removed with the hline command.

```
# customer totals
hline "---",erase
# example of UnForm command with continuation to next line
box "--------------------------------------------------", \
     -1,.25,{RIGHTCOL-53},1.25
bold "--------------------------------------------------",0,1,120,1
```

Finally, grand total lines are treated with special fonting and a box.

```
# grand totals
const DFONT=cgtimes,11,bold
# sample of box command with increased thickness and double lines
box "Grand Total:",-9.5,-1.25,MAXRCOLS,2.25,5,30,dbl 9
font "Grand Total:",0,0,12,1,BDFONT,13
font "Grand Total:",{scol-10},0,{colw},1,DFONT,decimal
font "Grand Total:",{scol-10+1*(colw+1)},0,{colw},1,DFONT,decimal
font "Grand Total:",{scol-10+2*(colw+1)},0,{colw},1,DFONT,decimal
font "Grand Total:",{scol-10+3*(colw+1)},0,{colw},1,DFONT,decimal
font "Grand Total:",{scol-10+4*(colw+1)},0,{colw},1,DFONT,decimal
font "Grand Total:",{scol-10+5*(colw+1)},0,{colw},1,DFONT,decimal
font "Grand Total:",{scol-10+6*(colw+1)},0,{colw+1},1,DFONT,decimal
```

LABELS – text labels to laser labels (advanced.rul)

UnForm is capable reading rows of input, parsing those rows into logical pages, and reproducing the output with different dimensions. A typical situation that can take advantage of this is if your application is designed to print mailing labels on continuous label stock on dot matrix printers. The labels can be 1-up, 2-up, or any other dimensions. As long as each label has a consistent number of rows and columns, UnForm can parse each label and treat each label as a logical page with the across and down commands. To use this sample, you must add "-r labels" to the command line.

```
uf80c –i sample4.txt –f advanced.rul –r labels –p pdf –o client:labels.pdf
```

This statement header identifies the rule set. The name is used in the –r command line option.

```
[labels]
```

Each label "page" is 35 columns and 6 rows of input text. If each line is 106 to 140 characters wide, then four labels are parsed from the columns. When the output is produced, each label will be 30 columns by 6 rows. The labels will be arranged 3 rows across and 10 down the page. UnForm will actually print 3x30=90 columns and 10x6=60 rows on each physical page.

Most laser label stock has ½ inch top and bottom margins. The margin command adds 75 dots (¼ inch) to the standard UnForm top and bottom margins, which default to ¼ inch.

In this sample, the text of the labels is printed from lines 1 to 4. By using the vshift 1 command, UnForm will move the text to lines 2 through 5. The shift command moves the text to the right.

```
page 35,6
rows 6
cols 30
across 3
down 10
font 1,1,40,6,cgtimes,12
margin 0,0,75,75
vshift 1
shift 2
# manual feed tray is usually 2
# tray 2
```

The barcode command supports both 5 and 9-digit formats of the postnet barcode. To get either to print, the prepage code block sets one or the other variable (zip$ or zip9$), and both commands are issued. A null value is not barcoded. The prepage code extracts the zip code from line 3 or 4 of the label. It then determines the length and sets zip$ or zip9$ appropriately.

```
barcode 2,6,{zip$},900,11.0,2
barcode 2,6,{zip9$},905,11.0,2

prepage{
# get zip code from line 3 or 4
zip$="",zip9$="",zipline$=""
if trim(text$[4])>"" then zipline$=trim(text$[4])
if zipline$="" then if trim(text$[3])>"" then zipline$=trim(text$[3])
while zipline$>""
     x=mask(zipline$,"[0-9][0-9][0-9][0-9][0-9]")
     if x>0 zip$=zipline$(x)
     zipline$=""
wend
# remove possible hyphen and validate length
x=pos("-"=zip$); if x=6 then zip$=zip$(1,5)+zip$(7)
if len(zip$)<>5 and len(zip$)<>9 then zip$=""
if len(zip$)=9 then zip9$=zip$,zip$=""
}
```

132x4 – multi-up, scaled reporting (advanced.rul)

This sample rule set will work with any 132 column by 66 row report. To use it, you must add "-r 132x4" to the command line. The report uses the across and down commands to scale the report to print four logical pages to a physical page.

uf80c –i sample3.txt –f advanced.rul –r 132x4 –p pdf –o client:132x4.pdf

The rule set header identifies the name.

```
[132x4]
```

The page dimensions are defined as 132 columns by 66 rows. UnForm will scale each page to fit 2 across and 2 down on a physical page (264 columns and 132 rows). The report is printed in landscape orientation. A box is drawn around each page, and the hline command will convert all occurrences of 3 or more dashes to horizontal lines.

```
cols 132
rows 66
across 2
down 2
landscape
cbox .5,.5,132.5,66.5
hline "---"
```

ZEBRA LABEL – Zebra® label printer example (advanced.rul)

UnForm offers an optional Zebra printer driver, which produces ZPLII code. Within the limits of the ZPL language, UnForm produces enhanced forms for Zebra printers in much the same way it does for laser printers. Some key differences are: fonts are identified differently and are limited in scalability, shading is either 100% (black) or 0% (white), and the **barcode** command is more extensive and capable than the laser printer **barcode** command.

When executing a Zebra run, it is critical to tell UnForm how large the labels are. This is done with a special syntax on the "-paper" command line option. Also, UnForm needs to know what print density is used by the printer. This is determined by the "-p zebra*n*" option, where *n* is either 6, 8, or 12 dots per millimeter. You may need to adjust this sample command line to match your Zebra printer, as it assumes an 8 dpmm printer and 3.25 by 5.5 inch label stock.

uf80c –i samplez.txt –f advanced.rul –p zebra8 –paper 3.25x5.5 –o *zebra-device*

This label is scaled to 40 columns and 35 rows.

```
[zebra label]
detect 0,1,"Zebra Barcode"
cols 40
rows 35
```

The prepage code block gets the PO number, setting it into a variable po$, and removing the PO number from the text with a set() function.

```
prepage{
po$=""
po$=cvs(get(2,16,10),3)
trash$=set(2,16,10,"")
}
```

The From and To sections draw boxes, change fonts, and re-allocate the lines of text from row 10 to row 14 with a series of text commands followed by an erase command.

```
# From section
box 1,1,39,8,3
text 2,2,"From:",font A
font 2,3,35,6,font 0,9

# To section
box 1,9.75,39,10.5,5
#text 2,10.6,"To:",font 0
```

```
text 3,11,{get(2,11,30)},font 0,12
text 3,12.25,{get(2,12,30)},font 0,12
text 3,13.5,{get(2,13,30)},font 0,12
text 3,14.75,{get(2,14,30)},font 0,12
text 3,16,{get(2,15,10)},font 0,12
erase 2,11,30,5
```

This group of commands prints three different barcodes on the label. First, a postnet code is printed from the zip code located at column 2, row 15, for up to 10 characters. Then a UPS maxicode barcode is printed with SDSI's address. Last, a "3 of 9" barcode is printed using the variable po$, derived in the prepage{} code block above.

```
# bar codes
barcode 10,18.25,{trim(get(2,15,10))},Z,33

text 2,24,"Maxicode",font 0,10
barcode 2,25,{"999840956820000" + $0a$ + "SDSI"+ $0a$ + "2195 Talon
Drive" + $0a$ + "Latrobe, CA 95682"},D

box 17,25,22,12,3
text 18,25.75,"Our PO No (in code 39):",font A,21
barcode 20,28,{po$},3,120,2,text above
```

PDF Outline Sample (advanced.rul)

UnForm supports PDF outlines (or bookmarks) when using the pdf driver. Outlines can be multiple levels, and each outline tree can be different levels deep. UnForm assumes each outline branch points to a page. To control the text shown in the outline, you set the variable outline$ in a prepage or precopy code block. This variable is parsed as each page is printed. Multi-level entries are created by delimiting the text of the levels with a vertical bar (|) within the contents of the variable.

The file sample5.txt contains the contents of a 14-page report featuring two sort and subtotal levels, as well as grand totals and a recap page. The outline tree for this report will be based on the salesperson (outer sort) and class code (inner sort), along with specific page entries for the report total and recap page. As there are no detect statements, you need to specify the –r option on the command line, as shown.

uf80c –i sample5.txt –f advanced.rul –r outline –p pdf –o client:outline.pdf

```
[outline]
```

Set the page dimensions and turn on the outline feature with the outline keyword. The default outline title for each page is simply "Page n", but a code block can override the outline text by setting the variable outline$.

```
cols 132
rows 66
outline
```

The prepage code block looks on each page for the following cases, in order:

- *A 3-digit salesperson number at the first column on line 7*
- *A salesperson subtotal heading on line 8*
- *A report total heading on line 8*
- *A recap page heading on line 2*

For the first two types of pages, a two level outline entry is created (level 1|level 2 structure). For the report total and recap pages, a single level outline entry is created.

```
prepage{
# default outline setting matches prior page
outline$=lastoutline$

# if line 7 starts with 3 digits, set 2-level outline slsp+class
if mask(get(1,7,3),"[0-9][0-9][0-9]") then \
 outline$="Slsp "+get(1,7,3)+"|Class "+get(13,7,2)

# if line 8 contains this, it is a salesperson subtotal
if pos("SALESPERSON: "=text$[8])>0 then \
 outline$="Slsp "+get(14,8,3)+"|Totals"
# if line 8 contains this, it is a report title
if pos("*Report"=text$[8])>0 then \
 outline$="Report Total"

# if line 2 contains this, it is the recap page
if pos("RECAP PAGE"=text$[2])>0 then \
 outline$="Recap Page"

lastoutline$=outline$

}
```

Additional Sample Rule Files

The following table describes several other sample rule files that have been designed to demonstrate specific techniques. There are two types of rule files provided:

- Cmd – adds to the documentation for specific commands
- Tool – helps an integrator to add new features to their own rule files

Rule file name	**Commands**	**Functions**	**Variables**	**Comments**
SampleCmdAFO.rul		gtextcount, gtextitem, gtextfind		How to work with Application Formatted Output
SampleCmdBarcode.rul	barcode	exec		Includes relative barcode
SampleCmdBox.rul	box, cbox	exec		
SampleCmdCircle.rul	circle	inchtocols		
SampleCmdConst.rul	const, global, local			
SampleCmdCopy.rul	copies, pcopies, if copy/end if, attach		Copy	Mult-copy, multi-format, variable number of copies
SampleCmdDetect.rul	detect			Many examples with explanations
SampleCmdDuplex.rul	duplex			
SampleCmdErase.rul	erase, notext, hline, vline	sub		
SampleCmdFont.rul	font, cfont			
SampleCmdImage.rul	image, attach			image justification
SampleCmdImages.rul	images			1 sample for archived images
SampleCmdLine.rul	line, hline, vline	exec		nice use of multiple cmds within 1 exec. Saves time.
SampleCmdMicr.rul	micr			
SampleCmdMove.rul	move, cmove			Includes relative moves
SampleCmdShade.rul	shade, cshade	exec		Includes relative shading and greenbar look
SampleCmdText.rul	text			Shaded, wrapped, fit,

				rotated, justified
SampleCmdTrayBin.rul	tray, bin			

Rule file name	**Commands**	**Functions**	**Variables**	**Comments**
SampleToolArchive.rul	archive			will do logging with Base Logging below
SampleToolChkContinued.rul	text	exec, get	pagenum, uf.maxpage	look ahead with get
SampleToolDispAFOText.rul	text			AFO field positions
SampleToolDocTitle.rul	text			
SampleToolFullBox.rul	cbox			
SampleToolLogging.rul		log		Base Logging
SampleToolLogo.rul	image			can use justification
SampleToolMultCpy.rul	copies, pcopies			
SampleToolPageSplitter.rul		getpage, putpage, delpage		
SampleToolPageXofY.rul	text	exec,get	pagenum, uf.maxpage	look ahead with get
SampleToolPostnet.rul	barcode	exec,mid		
SampleToolPrint.rul			skip	logging with Base Logging above
SampleToolScanBcd.rul	barcode	exec		barcode
SampleToolWatermark.rul	text	exec		shaded text

PROGRAMMING CODE BLOCKS

The prejob, predevice, prepage, and precopy subroutines (and their associated post*xxx* routines) open the world of Business Basic programming to the report and form designs. With a full programming language at your disposal, it is possible to customize and manipulate the forms, and to interact with other applications and devices, or with the operating system.

An experienced BBx or ProvideX programmer (ProvideX is the actual dialect used, with lexical compatibility added for most BBx syntax) typically performs the programming of these subroutines. However, programmers experienced in other languages, particularly other dialects of Basic, can easily learn the fundamentals of Business Basic and perform these programming tasks. Several of the sample forms include some programming, and there is a complete reference guide available from the ProvideX web site: www.pvxplus.com. In this manual, we have provided some basic (no pun intended) information that will assist developers experienced in other programming environments.

It should be noted that an UnForm job is not a stand-alone pvx program. Instead, it is a combination of a an UnForm job wrapper written in static pvx code, and user-defined code blocks, which are executed at particular times as subroutines while the job progresses. Therefore, code blocks act as subroutines and not full pvx programs. In addition, some pvx syntax has been overridden to ensure compatibility with previous versions of UnForm, and many UnForm-specific functions have been added so that code blocks can perform UnForm-specific tasks. Lastly, the user-interface features of pvx are not available, as UnForm jobs run in background and are not connected to a user screen.

Basic Syntax

Statements
A statement consists of a single verb and any arguments or parameters suitable for that verb. Multiple statements can be placed on a single line by separating them with a semicolon (;). Statements can be preceded by a label, which consists of a label name followed by a colon. Label names must follow the same naming conventions as numeric variables.

Variables
There are two types of variables in Business Basic: string and numeric. Variables that end in a "$" character are treated as string variables. They can hold any amount of text data, limited only by system memory. Numeric variables can contain any number or integer. UnForm sets precision to 10, so that up to 10 digits to the right of the decimal are maintained accurately.

Variable names can be up to 31 letters, digits, and underscore characters, and must start with a letter. Variables can't start with "fn" and should not start with "uf".

work$, account01$, and cust_name$ are valid string variables.
cust-name$ is invalid.
amount, period_12, and six are valid numeric variables.

Arrays can be defined for both string and numeric variables. Arrays must be defined to a fixed number of elements with a DIM statement, and array elements can then be referenced as variables. Arrays can contain up to three dimensions.

dim amount[12] defines a 13-element array, a[0] … a[12].
dim x$[1:6,1:20] defines a 2-dimensional string array. The first dimension ranges from 1 to 6, the second from 1 to 20. x$[2,20] would be a valid element in this array.

The dim statement can also be used to initialize strings to a specified length. Dim a$(12), for example, will set a$ to 12 spaces.

There are special string constructs available in ProvideX. These are called string templates or composite strings. Details about these constructs can be found in the language manual for ProvideX, available from www.pvxplus.com.

Functions
Many functions are available in Business Basic. Most will be familiar to a Basic programmer. Functions consist of a word, an opening parenthesis, one or more arguments, and a closing parenthesis. The function returns a string or numeric result, which is typically used as part of an expression, or in an assignment. Wherever a string or numeric value can be used, a string or numeric function can be used. In addition to internal Business Basic functions, UnForm also provides some functions that perform tasks typical to print stream environment in which it runs.

String and numeric representation
Strings are made up of concatenated bytes. They can be represented as literals inside double quotes, such as "Name:", or as hexadecimal strings inside "$" delimiters, such as $1B45$ for Escape-E. They can also be made up of combinations of literals, hex strings, string variables, and functions that return string values. These values are combined using the "+" operator to concatenate each string together. For example, a string containing quotes could be constructed one of these ways: chr(34)+"some text"+chr(34); or 22+"some text"+22, or quote$+"some text"+quote$. Since chr(34) and 22 both represent a quote character, and it would be possible for the variable quote$ to contain the same, all these expressions can represent the same string.

Substrings can be derived from a string variable with the syntax *stringvar*(*start* [,*length*]). For example, if account$ is "01-567", then account$(4,3) would return the value "567". Substrings references with positions that aren't in the string result in errors, so care must be used. To avoid the possible errors, the mid() function can be used.

Numbers can be represented as integers or decimal numbers, or, like strings, can be represented as expressions containing literal numbers, numeric variables, and numeric functions. With numbers, there are more operators available to produce the expressions. A literal number is just a series of digits, with an optional decimal point and an optional leading minus sign. 1995.99 and -100.433 are valid numbers. Other punctuation, such as thousands separators or currency symbols, are invalid in a number though they can be added when a number is formatted as a string for output.

Operators
Business Basic has the following standard operators:

+	concatenate strings or add numbers, depending on context
-	subtraction
*	multiplication
/	division
^	exponentiation
=	testing for equality, or assignment, depending on context
>	testing for greater than
>=	testing for greater than or equal to
<	testing for less than
<=	testing for less than or equal to
<>	testing for inequality
()	controlling precedence
and	combining expressions with logical "and" in conditions
or	combining expressions with "or" in conditions

+=	appends right-side value to a string or adds to a number
-=	subtracts right-side value from a number
*=	multiplies a number by right-side number
/=	divides a number by right-side number

++var, *var++*, *--var*, *var++* increments or decrements a numeric variable by 1, either before or after an operation, depending on the position of the ++ or – operators.

If Then Else

The structure of IF…THEN…ELSE statements are simple and unblocked. The IF must be followed by an expression to test. The expression can be simple or complex, and must resolve to a single Boolean or numeric result. For numeric results, a 0 is considered false, and anything else is considered true. Once resolved, if true the THEN clause is executed, otherwise the ELSE clause, if present, is executed.

Both the THEN clause and the ELSE clause can contain any statements, including nested IF statements. A closing END_IF after a THEN or ELSE clause will terminate the conditional nature of statements following it.

Here are some examples of IF statements:

```
if amount < 0 then text$="Credit Balance"
if x$="A" then desc$="Acme Rental" else if x$="S" then desc$="Smith & Sons" else desc$="N/A"
if testmode then dummy$=set(1,1,10,"Test Mode") end_if; goto exitsub
```

UnForm's code block parser also supports blocked if-then-else syntax, like this (optional elements indicated in square brackets):

```
    If condition [then]:
        Statement1
        Statement2
    [else
        Statement1
        Statement2
        … ]
    End if
```

The key elements are the colon (:) at the end of the "if *condition*" line and the closing "end if". The structures can be nested, with additional if statements inside the if or else sections.

While Wend Loops

One of Business Basic's looping structures is the WHILE..WEND loop. At the top of the loop is a while *condition* statement, where the condition is evaluated like an IF clause. As long as the condition is true, or returns a non-zero value, the statements up until the closing wend statement are repeated. To escape the loop, you can use the BREAK verb, the EXITTO label verb, or set variables such that the condition is false before executing the wend verb. To iterate the loop from within, use the CONTINUE verb.

Here is a simple WHILE…WEND syntax that substitutes (") with (') in a string:

```
x=pos($22$=work$)
while x > 0
        work$(x,1)=""
        x=pos($22$=work$)
wend
```

For Next Loops
Another commonly used loop structure is the FOR…NEXT loop. A FOR statement identifies a variable, a start value, an end value, and an optional step value. The variable is set to the start value; the loop statements are executed until a NEXT statement is encountered; the variable is incremented by the step value; and, until the end value is exceeded, the loop statements are repeated. To exit the loop before the end value is reached, use the BREAK verb or the EXITTO label verb. To iterate the loop from within, use the CONTINUE verb. Here is an example that would perform the same substitution shown above (though more slowly):

```
for i=1 to len(work$)
        if work$(i,1)=$22$ then work$(i,1)=""
next i
```

File Handling
Business Basic has very powerful facilities for handling files. Not only are there intrinsic keyed file types, but also text files and pipes can be used.

If the application with which UnForm is integrated is written in ProvideX, then full native access to the data files is available. If the application is written in BBx, then the bbxread() function can be used to obtain record data via an instance of BBx.

If UnForm is working with a non-Business Basic application (e.g. C, Cobol, Informix, Oracle, etc.), there are additional means to obtain data, via ODBC on Windows or pipes on UNIX.

<u>Opening Files</u>

File access is performed through an open file channel. The OPEN statement opens the file on a numeric channel in preparation for later file access. Open(99)"customers.dat" opens the named file on channel 99. Channel numbers can range from 1 to 32767, though the operating system will typically impose a limit on the number of simultaneous channels that can be opened. Channel numbers must be unique. Once opened, that channel number is no longer available until closed. To avoid conflicts with channel numbers, it is common to use a special function that returns an available channel number, UNT. Here is a typical syntax:

```
cust=unt
open(cust)"customers.dat"
```

After that, file access verbs can use the cust variable to access the "customers.dat" file.

To open a pipe channel, you could do the following:

```
faxlist=unt
open(faxlist)"|sqlexec 'select cust,faxnum from customers'"
read(faxlist)line1$

labelprt=unt
open(labelprt)">lp –dlabels"
print(labelprt)"To: "+name$
print(labelprt)"    "+address1$
```

Reading Files

There are two verbs used for reading channels: READ, and READ RECORD. The READ verb understands line and field separators, whereas the READ RECORD verb reads blocks of a specified size or whole records, in the case of intrinsic keyed file types. The READ verbs accept several options, including "key=*string*", "ind=*index*", "err=*linelabel*", "end=*linelabel*", and others. Full details can be found in the language reference manuals. Labels can be actual labels in code (*label*:), or a symbolic label, such as *next, *break, or *continue. UnForm also recognizes err=next as a synonym for err=*next.

To read from an intrinsic keyed file (ProvideX files only), you might use one of these:

```
read(cust,key=custkey$,err=next)*,name$,*,*,*,*,faxnum$

read record(cust,key=custkey$,err=next)custrec$
name$=custrec$(7,30),faxnum$=custrec$(112,10)
```

To read from a pipe or a text file, you may not use a key= clause, so you just read sequentially through the file:

```
read(faxlist,end=done)cust$,faxnum$
```

Writing files

You probably would not want to write to your application files, but you may well want to write to external devices or log files. Writing is performed with these verbs: WRITE or WRITE RECORD and PRINT. Each uses a channel number and arguments to print. PRINT terminate its values with a line-feed character ($0A$), unless a comma follows the last argument. WRITE RECORD will write a single string variable without any termination so it is suitable for binary or blocked output. WRITE terminates values with an internal field separator, normally $8A$, which is not useful when writing files that will be used by other applications, but which is recognized by READ.

```
print (logfile)"Customer: "+custname$+" printed on "+date(0,tim:"%D-%M-%Y:%Hz:%mz")
dim block$(128); block$(1)=custname$,block$(31)=str(amount:"000000.00"); write record(log)block$
```

Object Oriented Programming

UnForm supports programming with objects, which are self-contained programming units (called objects) that have data elements (known as properties) and functions (known as methods). Object oriented programming (commonly referred to as OOPS) is a modern technique that has become popular with the rise of languages such as Java, C#, C++, and VB.NET. The definition of an object, that is, its properties and methods, is encapsulated within a program unit called a "class". The terms "class" and "object" are sometimes used interchangeably, but there is a distinction: a class is a description or definition, and an object is a physical, programmable representation of a class.

There are many built-in objects supplied with UnForm, which can be created (or instantiated) and used within code blocks as needed. Additional custom objects can be created as a professional service.

Object Instantiation

Code blocks can create new objects based on a class with the new() function. Each object is a simple numeric variable. The new() function returns an object ID number, and all action on that object is processed through the numeric variable.

There can be any number of objects in memory at a given time. Each object that is created is unique, with its own data.

The syntax of the new() function is:

***objvar*=new("*classname*"[,*arg1*[$],…][,err=*label*|*next)**

When this function is executed in a code block, a new object is created and assigned to *objvar*. Some objects accept arguments during this initialization step, and the arguments are provided in a comma-separated list after the class name.

If an error occurs during the object creation, the object is not created, and the err=*label* is executed.

Object Access

Once the object is created, *objvar* can be used to reference that object's properties and functions, using the apostrophe (') operator (-> is a synonym for '):

> *objvar*'*property*[$]=*var*[$] | *number* | "*string*" | *expression*
> *var*[$]=*objvar*->*property*[$]
>
> *objvar*->*function*(*arg1*[$][,…])
> *var*[$]=*objvar*->*function*(*arg1*[$][,…])

Functions can be accessed as functions, which return a value, or as subroutines, which simply execute the function code.

Object Destruction

To destroy an object, use the 'drop object' statement. Since the object variable no longer references an object, it can be set to 0, reused, or ignored. Once an object is no longer needed, it is good practice to destroy it. For example, beware of objects created repeatedly in loops and not destroyed.

drop object *objvar*

Built In Objects

The following objects are provided with UnForm and can be instantiated with the new() function. Property and method references follow.

- addrbook - address book management
- binfile - binary file access
- collection - collections of values by index and name
- date - date management
- doclist - library document lists
- http - http/https client for interacting with web servers
- json – conversion of data to JSON format for ease of Javascript processing
- inifile - ini file access
- keyfile - keyed file access
- libraries - library lists
- library - library management
- marked – marked record management
- rac - remote access codes for documents
- search - library search execution
- system - operating system and file system access
- textfile - text file access
- webapi – creation of web-oriented URL strings for web form and DTC processing
- xmlreader - xml document parsing

Examples of using many of these objects can be found in the rule file samples/objects.rul.

addrbook

book=new("addrbook",name$[,create])

Address book object stores delivery records, identified by an entity ID and document type. Each record is maintained as a string template with the following fields:

entityid$, doctype$, entityname$, contactname$, sendto$, combine.

All address books are stored in the addrbks subdirectory under the UnForm server. The files are named as name$+".dat", so the name must be a valid file name for the operating system. If the address book exists, it is opened for use. If not, and the create flag is true (1), it is created. An error occurs if the address book is not found and the create flag is missing or false (0).

Properties

filename$ is a read-only property that contains the actual disk file used to store the address book records.
**template$** is a read-only property that can be used to dimension an address entry template. i.e. dim rec$:book'template$.

Methods

count() returns the number of entries in the addressbook.
deladdress(entityid$,doctype$) removes the address book entry. Returns 1 or 0.
getaddress(entityid$,doctype$,address$) fills address$ template with the requested entry, returns 1 if successful, or 0 for failure. For example, if there is no record found, a 0 is returned, and address$ will be an empty, but dimensioned, template string.
newaddress(address$) creates an empty address$ template that can be filled by a code block before writing. Always returns 0.
putaddress(entityid$,doctype$,address$) updates address book using entityid$ and doctype$ identification, writing the data in the template address$. Returns 1 if successful, or 0 if not. Note that address.entityid$ and address.doctype$ are set to entityid$ and doctype$.
range$(start,count[,order[,descending]]) returns a LF-delimited list of TAB-delimited records. Each record contains the same fields as the template above, separated by a TAB character. If start or end are 0, they are the first and last records, respectively. The optional order value can control the sequence of the records: 0 for entity ID/DocType, 1 for entity name, 2 for contact name, or 3 for send to address. The optional descending value can be 0 for ascending sequence, 1 for descending sequence.

binfile

fl=new("binfile"[,filename$])

The binfile object provides read/write access to the file in binary fashion, where there is no concept of a record. All access is to specific byte positions in the file, or the file as a whole. The filename$ argument specifies what file to open or create. If no file name is supplied, then a temporary file will be created. This temporary file will be erased when the UnForm job is complete.

Properties

filename$ is a read-only property that contains the full path to the file name opened or created.
size contains the file size in bytes. If size is assigned, the file is expanded or truncated to the specified size.

Methods

append(block$) appends block$ to the end of the file, and returns the size of the file.
delblock(index,length) removes the specific bytes from the file, and returns the size of the file.
getblock$(index,length) returns file content at the specified position and length. The position is 1-based, so the first byte is 1.
getfile$() returns the entire contents of the file.
insblock(index,block$) inserts block$ in the file at the 1-based index position. Returns the size of the file.
purge() sets the file length to 0, just like setting the size property to 0.
putblock(index,block$) writes block$ to the file at the 1-based index position, replacing data at that position. Returns the size of the file.
putfile(block$) updates the file contents to block$, truncating or expanding as necessary. Returns the size of the file.

collection

obj=new("collection"[,noclose])

The collection object provides access to a collection of items stored by a key or index. Keys can be up to 127 characters long. Methods are provided to add, update, and remove elements from the collection, to find items in the collection, and to list items in the collection. Collection storage is mapped to disk, so there can be a virtually unlimited number of elements. If you wish to have the disk file remain open for the lifetime of the object, supply the noclose argument as true (non-zero). Note however, that if many collections are maintained during a job, only a limited number can have the noclose option set to true, since operating systems impose a limit on how many files can be open at one time by a process.

Properties

count is a read-only property that holds the number of items in the collection.

Methods

add(ky$,value$|value) adds a string or numeric value to the collection, identified by unique key. An error occurs if the collection already contains the specified key.
add(value$|value) adds the specified string or numeric value at a sequential index position. No key is associated with the item.
addlist(values$[,dlm$]) adds several string values identified by sequential index. The list of values is delimited by a linefeed character ($0A$), or by the delimiter if supplied.
clear() removes all items from the collection.
copyto(colobject) copies all elements from the this collection to the specified collection object.
exists(ky$) returns true (1) if ky$ exists in the collection, false (0) if not.
getitems$([dlm$]) returns a delimited list of values from the collection, using the specified delimiter. If no delimiter is specified, a linefeed ($0A$) is used.
getkeys$([dlm$][,order]) returns a delimited list of keys from the collection, using the specified delimiter. If no delimiter is supplied, then a linefeed ($0A$) is used. If order is 0, or not supplied, the keys are returned in the order added to the collection. The keys are returned in ascending sequence if order=1, or descending sequence if order=2.
item$(ky$|index) returns the string value identified by the key or sequential index.
item(ky$|index) returns the numeric value identified by the key or sequential index.
key$(index) returns the key of the item at the specified sequential index position.
keycur$() returns the current key, based on the last key operation.
keyfirst$() returns the first key in the collection (in ascending key sequence order).
keylast$() returns the last key in the collection.
keynext$() returns the next key in the collection, relative to the key of the last key operation.
keyprev$() returns the previous key in the collection, relative to the key of the last key operation.
remove(ky$|index) removes the item identified by its key or sequential index from the collection.
update(ky$,value$|value) updates the item identified by ky$ with value$ or value. Adds the key if it doesn't exist.

date

obj=new("date")

The date object provides date-oriented functionality, including date parsing and formatting. When the date object is created, the datetime property is set to the current date and time. A datetime value is a numeric Julian number, indicating the number of days since January 1, 4713 BC, plus time expressed as a fraction of a day. Methods are provided for parsing a text date into a datetime value, to format a datetime value into a human-readable value, and to calculate elapse time between two datetime values in different increments.

Properties

d is a read-only property that provides the day.
datetime is the date and time value upon which all methods work. The datetime value is initially set to the date and time at the moment the object is created. It can be updated to the current date and time with the update() method, or set to a value via the parsedate() function or setdate() function.
hr is a read-only property that provides the hour (using a 24 hour clock).
m is a read-only property that provides the month.
mn is a read-only property that provides the minute.
se is a read-only property that provides the second.
utcoffset provides the offset from Universal Time for the local time zone. The value is provided as a fraction of a day, so it can be added or subtracted from datetime without any conversion.
y is a read-only property that provides the year.

Methods

days([enddatetime]) returns the number of hours between the current date and time and datetime, or the supplied date and time (such as the datetime value of another date object).
format$([fmt$]) returns the formatted date and time. If no format is supplied, or it is null or "utc", the date is returned in UTC format in adjusted by the utcoffset property. For example: Fri, 7 Aug 2009 23:17:04 +0000. If fmt$ is "local", then the format is the same, but reported for the local time zone. For example: Fri, 7 Aug 2009 16:17:04 -0700. If fmt$ is "ymd", then a 14-byte string is returned in the format yyyymmddhhmmss. Additional formats are custom, using mapping characters that are replaced with appropriate date/time components. The characters are: • am or pm • AM or PM • YYYY or YY (4- or 2-digit year) • MMMM, MMM or MM (month name, abbreviation, or number) • DDDD, DDD, or DD (day name, abbreviation, or number) • HH (24 hour clock) • hh (12 hour clock)

• mm • ss Other characters represent themselves. "MM/DD/YYYY" would return a typical US date. DD/MM/YYYY would return a typical Canadian date.
hours([enddatetime]) returns the number of hours between the current date and time and datetime, or the supplied date and time.
minutes([enddatetime]) returns the number of minutes between the current date and time and datetime or the supplied date and time.
parsedate(datestr$[,fmt$]) parses a human readable date using fmt$ for parsing rules, sets datetime, and returns datetime. If not supplied, the default format is "utc". The parsing rules are "utc" for UTC format, "ymd" for yyyymmddhhmmss format, or "mdy" or "dmy" for delimited dates, such as 12/31/2009 or 31/12/2009.
seconds([enddatetime]) returns the number of seconds between the current date and time and datetime, or the supplied date and time.
setdate(year, month, day [,hours [,minutes [,seconds]]]) sets the date and time according to the arguments provided, and returns datetime. Only year, month, and day arguments are required.
update() updates datetime to the current date and time, and returns datetime.

doclist

obj=new("doclist"[,filename$[,initfile]])

A doclist is a list of documents from one or more libraries. Each document is identified by a library, document type, and document ID, called a document record. Doclist objects can be manipulated and listed. A doclist object is also created by the search object, providing a set of methods for processing the search results.

If filename$ is supplied, then that file is opened and used for the document lists. If not, a new temporary file is created, which is automatically erased when the object is destroyed. If initfile is provided and true (1), the document list is initialized so no document records are present.

Properties

listfile$ is a read-only property that provides the name of the document list data file.

Methods

clear() clears the list of all documents.
count() returns the number of records in the list.
count(library$) returns the number of records in the list for the given library.
count(library,doctype$) returns the number of records in the list for the given library and document type.
deldoc(library$,doctype$,docid$) removes the specified document from the list. Note this does not affect the document stored in the library.
getdoc(library$,doctype$,docid$[,prop$]) returns 1 if the document exists in the list. If prop$ is provided, it creates it as a template and fills it if the document exists. The prop$ template contains document properties of the document itself, from the archive storage system, using a library object. The following properties are provided: • prop.date$ • prop.time$ • prop.title$ • prop.entityid$ • prop.notes$ • prop.keywords$ • prop.categories$ • prop.links$
getdocs$(library$,doctype$,first$|first,count [,descending]) returns a list of document library, doc type, and doc ID's. The three fields are delimited by tabs ($09$) and the records are delimited by linefeeds. Within the range of library and document type, up to count records are returned, from starting with the document ID indicated by first$, or index indicated by first. If descending is true (1), the list is returned in reverse order.
getlibs$() returns a linefeed ($0A$) delimited list of libraries in the list.

gettypes$(library$) returns a linefeed delimited list of document types, within a library name, in the list.
movefirst(library$,doctype$,docid$[,docprop$]) moves to the first record and returns the library, document type, document ID, and optional document properties template with the following fields: date$, time$, title$, entityid$, notes$, keywords$, categories$, and links$, referenced as docprop.*name*. Returns 1 if successful, 0 if not.
movelast(library$,doctype$,docid$[,docprop$]) moves to the last record. Returns 1 if successful, 0 if not.
movenext(library$,doctype$,docid$[,docprop$]) moves to the next record. Returns 1 if successful, 0 if not.
moveprev(library$,doctype$,docid$[,docprop$]) moves to the previous record. Returns 1 if successful, 0 if not.
moveto(library$,doctype$,docid$) navigates a specific point in the document list. Use this as a seed for subsequent moveprev() or movenext() methods.
putdoc(library$,doctype$,docid$) adds the specified document to the list. Note this does not affect the document stored in the library.

http

obj=new("http"[,url$])

The http object provides access to HTTP servers, more commonly known as web servers. HTTP is the protocol used by web browsers to communicate with web servers. The http object performs the client-side activity of the HTTP protocol, connecting to and requesting actions of a web server. To use the object, set the url or any of its component element properties, optionally add cookies or files to be uploaded, and optionally set the user and password if authentication is required. Then issue a getrequest() method to submit the request and obtain a response.

Properties

header$ contains the last set of headers from the server.
host$ is the hostname, or IP address, of the HTTP server.
method$ sets the server communication method. It must be "get" or "post". To simulate a form or file upload, use the post method. To simulate a link click in a browser, use the get method. Generally, if you expect to send a large amount of data, use post.
password$ can be specified if the server requires authentication.
path$ is the portion of the url after the hostname:port and before the ?query string in. This normally represents a name-mapped file or script on the server.
port is the port on which the server is listening. The default ports are 80 for http protocol, and 443 for https protocol, but these values can be configured on the server and may vary.
protocol$ should be http or https (for a secure server connection).
query$ is the trailing portion of the url which scripts interpret for variable data. The data follows the path and a "?" delimiter, and is often in name=value pairs. The query string must be URL-encoded. The query string can be generated using the addfield() method, or you can manipulate the string directly using the urlencode() function.
reason$ contains the last reason text from the server.
response$ contains the last response body from the server.
status contains the last status code from the server.
timeout establishes the timeout for server communication, in seconds. By default, there is no timeout, and the object will wait forever for a server response.
url$ is the full address of the currently requested web page. It is represented with a protocol (http/https), a host name, a port, path, and query string, such as "http://unform.com/sdsi.cgi?p=unform8".
userid$ can be specified if the server requires authentication.

Methods

addcookie(name$,value$) adds a set-cookie header to the http request headers. Some server applications require receipt of a cookie value.
addfield(name$,value$) adds a name=value pair to the query string, with proper url-encoding.
addfile(name$,filename$) adds a file to the submission
getfields() returns a count of the number of query string fields.
getheader$(name$) returns the value of the named header.
getname$(n) returns the name of the nth query string value.

getresponse([response$][,headers$][,status]) submits the request and fills response$, headers$, and status with the server's response. The response$ value contains the actual content, often an HTML page or XML document. The headers$ field contains linefeed ($0A$) delimited rows of 'name: value' pairs. The status code is the HTTP status of the response, a number whose meaning is defined in the HTTP protocol specification. For example, a status of 200 means OK, and a status of 404 means the requested file wasn't found.
getvalue$(n) returns the value of the nth query string value.

inifile

obj=new("inifile"[,filename$])

This class manages sections and items in a text file in "ini" format, where sections in the file have a [*name*] header, and items are stored in *name=value* lines. The ini file format is common in many applications, including UnForm. Both the uf80d.ini and ufparam.txt file use this format.

If filename$ is not provided, a temporary file is created that is erased when the object is destroyed.

Properties

content$ is the read-only content of the entire file.
filename$ is the read-only name of the file being managed.

Methods

getitem$(section$,name$) returns the value of the named line in the named section.
getsection$(section$) returns all lines in the section, delimited by linefeeds ($0A$).
putitem(section$,name$,value$) replaces or adds a line in the section, in name=value format.
putsection(section$,lines$) replaces or adds an entire section with the contents of lines$, which should be a series of name=value lines separated by linefeeds ($0A$).
removeitem(section$,name$) removes the line identified by name$ from section$
removesection(section$) removes the entire section.

json

obj=new("json")

The json object provides functionality to convert data from different file structures often found in UnForm server operations into JSON format used often in Javascript code. The common use for this function is for custom web form development, where data from a server-run rule set can be obtained in Javascript at runtime using the runRuleSet function found in common.js and available in the browser interface to document management archives.

All text is assumed to be encoded as ISO-8859-1.

Methods

fromdlm$(*fileorstring$* [,*delim$* [,*quotes* [,*header$*]]]) returns a JSON representation of a delimited file, such as a CSV file or tab-delimited file. If no delim$ character is provided, a tab ($09$) character is assumed. Quotes can be 0 or 1; 1 indicates that data that might contain the delimiter will be quoted. The first line of the file is assumed to contain column header names, and these names are used as the JSON object names (after replacing invalid name characters with underscores). Some delimited files do not contain a header row, in which case you can supply a delimited list of column names in header$. The response is an array of objects, where each file row is an array element, and each row is represented as an object with *name*:*value* pairs. If the file doesn't exist, the value of the argument is used as if it was file content.
fromcsv$(*fileorstring$*[,*header$*]) is equivalent to fromdlm$(*fileorstring$*,",",1,*header$*)
fromini$(*fileorstring$*) parses a file or string in INI format, with section headers in square brackets (i.e. [main]) and section data made up of lines of *name*=*value* pairs. Each section becomes an object whose value is another object.
fromtpl$(template$) returns a JSON object representing fields and data in a string template. For example, a library object can obtain properties of a document in a template: obj'getdoc(doctype$,docid$,tpl$), and the returned tpl$ value can be converted to JSON format using this method. Numeric and string types are maintained during this conversion.

A CSV to JSON example:

```
"First Name","Last Name",Age
"Joe","Smith",40
"Sue","Smothers",45
```

```
[ {First_Name:"Joe", Last_Name:"Smith", age:40} ,
  {First_Name:"Sue", Last_Name:"Smothers", age:45
]
```

An INI to JSON example:

```
[Section 1]
Name1=Value 1
Name2=Value 2
[Section 2]
Name1=Value 1
Name2=Value 2

{
	Section_1: {Name1="Value 1", Name2="Value 2" },
	Section_2: {Name1="Value 1", Name2="Value 2"}
}
```

keyfile

obj=new("keyfile"[,filename$])

A keyfile object provides access to a disk file that stores records based on a unique string key. Keys can be up to 127 characters long. Records can be of any size and format. If filename$ is not supplied, a temporary file is created that is erased when the object is destroyed.

Properties

chan is a read-only property that stores the underlying channel number that the file is open on.
count is a read-only property that contains the number of records in the keyfile.
filename$ is a read-only property that contains the name of the file being managed.
found is a read-only collection object handle that is filled by methods that find records or return ranges of records from the file.

Methods

delete(ky$) removes the record identify by ky$. If the key doesn't exist, an error occurs.
exists(ky$) returns true (1) if ky$ exists in the file, false(0) otherwise.
find(search$[,nocase[,invert]]) initializes the found collection object, then fills it with keys and records whose records contain the text search$. If nocase is true (1), then the search is case-insensitive. If invert is true (1), then records that do not contain search$ are added, rather than those that do. The function returns the number of records found.
findreg(regex$[,nocase[,invert]]) initializes the found collection object, then fills it with keys and records whose records match the regular expression regex$. If nocase is true (1), then the search is case-insensitive. If invert is true (1), then records that do not match regex$ are added, rather than those that do. The function returns the number of records found.
findwhere(whereexpr$,dlm$,quotes) initializes the found collection, then searches records that match the where expression. For this search, records are assumed to be in delimited format, with the supplied dlm$ value as the delimiter. If quotes is true (1), then fields are parsed assuming they may be quoted to protect delimiter values in field values. The structure of the where expression is of a Boolean expression using fields, values, comparison operators, parentheses, and AND or OR, using #*number* syntax to represent field numbers. All fields are assumed to be string data, but you can use num() to convert strings to numbers, so long as the string is an unpunctuated numeric value. Here are some examples: #2<>"" - field 2 not null #2="100" and (num(#3)>=0 and num(#3)<10000) - field 2 is "100" and field 3 is between 0 and 10000
keycur$() returns the key of the last accessed key accessed.
keyfirst$() returns the first key in the file in key sequence.
keylast$() returns the last key in the file in key sequence.
keynext$() returns the next key relative to the last key accessed.
keyprev$() returns the previous key relative to the last key accessed.

keyval$(recno) returns the key of the record number specified. The record number is a sequential value in key order.
range(first$,count[,descending]) initializes the found collection, then fills it with keys and records starting from the key first$, and adding up to count records. If descending is true (1), then records are returned in reverse key sequence. The number of records found is returned.
range(first,count[,descending]) initializes the found collection, then fills it with keys and records starting from the record number first, and adding up to count records. If descending is true (1), then records are returned in reverse key sequence. The number of records found is returned.
range(key1$,key2$) initializes the found collection, then fills it with keys and records from the key range provided. For example, to get all records with keys starting with "100", you might use range("100","100Z"). The two keys do not have to exist in the file. The number of records found is returned.
read$([ky$]|[recno]) returns a record. If a key or record number are supplied, the specified record is returned. Otherwise, the next record in ascending key sequence is returned, relative to the last record accessed. If the key or record number does not exist, an error is generated.
readlock$(ky$[,timeout]) reads and locks the record. If timeout is provided, then if the record is not available (locked by another task), the system will wait for up to timeout seconds before returning an error. If the key doesn't exist, or the record is locked, an error is generated.
write(ky$,rec$) writes the record identified by ky$. If the key already exists, the record is replaced. If not, it is added. If the key is locked by another task, an error occurs.

libraries

obj=new("libraries")

The libraries object provides access to the list of libraries that are known on a system. Lists are returned in linefeed-delimited ($0A$) format, and each list row can be in one of two formats: just library path names, or all library details. A template can be obtained that describes the library information that is returned, so it is possible to develop routines that parse a list and obtain specific library information. A name-only template contains one field: pathname$. A detail template contains the following fields, delimited by a tab ($09$) or other character, if specified:

- basename$ - the base filename of the library path.
- dirname$ - the directory portion of the library path.
- path$ - the full library path.
- category$ - the category, if any, of the library.
- description$ - the library description
- created$ - the library creation date, in yyyymmdd format
- defperms$ - default permissions (r, w, and/or d) in semicolon-delimited format
- forceseq - true (1) to force sequencers on sub IDs
- inactive - true (1) if the library is inactive an no longer can be updated
- count - the number of document records in the library

Methods

getall$([detail[,delim$]) returns the names of all libraries, including inactive ones. If detail is true (1), then lines of detail are returned; if not, only path names are returned. If delim$ is not null, that is used as a template delimiter character; if it is null, a tab ($09$) delimiter is used.
getdelete$(userid$,[detail[,delim$]) returns the names of all libraries the specified user can delete from. If detail is true (1), then lines of detail are returned; if not, only path names are returned. If delim$ is not null, that is used as a template delimiter character; if it is null, a tab ($09$) delimeter is used.
getnames$([detail[,delim$]) returns the names of all active libraries. If detail is true (1), then lines of detail are returned; if not, only path names are returned. If delim$ is not null, that is used as a template delimiter character; if it is null, a tab ($09$) delimeter is used.
getread$(userid$,[detail[,delim$]) returns the names of all libraries the specified user can read from. If detail is true (1), then lines of detail are returned; if not, only path names are returned. If delim$ is not null, that is used as a template delimiter character; if it is null, a tab ($09$) delimeter is used.
gettemplate$(detail[,delim$]) returns a string template definition for detail or non-detail return lines, optionally with the specified delimiter. If no delimiter is specified, a tab delimiter ($09$) is assumed.
getwrite$(userid$,[detail[,delim$]) returns the names of all libraries the specified user can write to. If detail is true (1), then lines of detail are returned; if not, only path names are returned. If delim$ is not null, that is used as a template delimiter character; if it is null, a tab ($09$) delimeter is used.

The following code fragment illustrates how one could parse a library list.

```
obj=new("libraries")
```

```
dim row$:obj'gettemplate$(1)
rows$=obj'getnames$(1)
x=pos($0a$=rows$)
while x>0
        row$=rows$(1,x-1)
        rows$=rows$(x+1)
        libname$=row.basename$
        libpath$=row.dirname$
        libcount=row.count
        x=pos($0a$=rows$)
wend
drop object obj
```

library

obj=new("library",libname$[,create])

The library object provides extensive access to the documents in a specific library, specified in libname$, including document listings and manipulation of individual document and sub-image properties, as well as general library information. If the libname$ provided is not found, a new library is created if the create field is true (1).

The library object enforces library security, and will generate appropriate errors if the active user login (specified with the setlogin() code block function) does not have rights to perform requested operations. Note that secure passwords can be configured in the browser interface and referenced in the setlogin() function.

Properties

All properties are read-only.

datecreated$ is the date the library was created, in yyyymmdd format.
forceseq is true (1) if sub IDs are auto-sequenced to prevent overwriting.
inactive is true (1) if the library is locked from further updates.
lasterrmsg$ contains the last known error message related to methods.
libname$ is the name supplied to the object.
pathname$ is the full system path to the library.
permission$ is the default permission setting for users that do not have specific permissions allocated. It is a semicolon-delimited list of "r", "w", or "d".
title$ is the library title, or description.

Methods

candelete(userid$) returns true (1) if the specified user can delete images or documents from the library.
canread(userid$) returns true (1) if the specified user can read the library.
canwrite(userid$) returns true (1) if the specified user can write to the library.
catcount() returns the number of category indexes in the library.
copydoc(doctype$,docid$,todoctype$,todocid$[,library$]) copies a document, indentified by doctype$ and docid$, including all its subdocuments, to the specified todoctype$ and todocid$. If library$ is specified, the document is copied to a different library. Returns 1 if successful. Returns 0, and fills lasterrmsg$, if not.
copysubdoc(doctype$,docid$,subid$,todoctype$,todocid$,tosubid$[,library$]) copies a subdocument, indentified by doctype$, docid$, and subid$, including its image data, to the specified todoctype$, todocid$, and tosubid$. If library$ is specified, the subdocument is copied to a different library. Returns 1 if successful. Returns 0, and fills lasterrmsg$, if not.
countcats([seed$]) returns the number of category indexes within the given seed. If no seed$ is provided, a count of initial segments is returned. If seed$ is provided, the count is the number of segments at the next level. Seed$ is pipe-delimited. For example, countcats("Customers|ByName")

would return the number of third-level segments within Customers|ByName.

countdocs(doctype$) returns the number of documents in the specified document type.

countdocsbycat(seed$) returns the number of documents for the specified category index, which is in a pipe-delimited segment series format.

countdocsbydate(date$) returns the number of documents for the specified date. Date$ should be in yyyymmdd format.

deldoc(doctype$,docid$) removes the specified document, and any of the document's sub IDs , from the library.

delsubdoc(doctype$,docid$,subid$) removes the specified subdocument from the library.

doccount() returns the number of documents in the library.

docexists(doctype$,docid$) returns true (1) if the document type and ID exist in the library, false (0) otherwise.

getcats$([seed$[,first|first$[,count[,descending]]]]) returns a linefeed delimited list of category indexes found in the library. Each index has pipe (|) segment delimiters. If seed$ is provided, the list is returned within the prefix found in seed$, which is a pipe-delimited list of segments. In this way, it is possible to get a list of indexes, sub-indexes, sub-sub-indexes, and so forth. The list starts at the record number specified by first, or the sub-segment value specified in first$. Up to count records are returned. If descending is true (1), the list produced in descending order.

getcats$() returns a list of initial category segments.

getcats$("Customers") returns a list of second segments within the "Customers" initial segment.

getcats$("Customers|ByName","B",1000) returns a list of up to 1000 category indexes within the Customers, ByName seed, listing values of the third segment, starting with "B".

getdates$([year[,month]]) returns a linefeed delimited list of document dates found in the library. Dates are returned in yyyymmdd format. If the year, or year and month, are provided, only dates for that period are returned.

getdoc(doctype$,docid$,doc$) fills the doc$ property template with document properties, and returns 1 if successful. The template contains the following fields:

- date$ (yyyymmdd)
- time$ (hhmmss)
- dateupdated$
- timeupdated$
- title$
- entityid$
- notes$
- keywords$
- categories$
- links$

getimage(doctype$,docid$,subid$,filename$) extracts the image data to the specified filename$. If filename$ is null, then a temporary file is generated with an appropriate extension and returned in

filename$. The temporary file is automatically deleted with the job is complete. A user-supplied file name is not deleted. Returns 1 if successful. If not successful, lasterrmsg$ will contain an error message.
getrange$(doctype$,first|first$,count[,descending[,delim$]]) returns a linefeed delimited list of document types and IDs within the specified document type. The list starts at the record number specified by first, or the document ID specified by first$. Up to count records are returned. If descending is true (1), then the list is returned in descending order. If delim$ is specified, the type and ID are delimited by that character. If no delim$ is specified, then they are delimited by a tab (09).
getrange$(first,count[,order[,descending[,delim$]]]) returns a linefeed ($0A$) delimited list of document types and IDs. The list range is in sequence based on the value of order: 0=type/ID, 1=date/time created, 2=title. The range is specified with first and count, indicating the starting document, and the maximum number to return. If descending is true (1), then the list is returned in descending order. If delim$ is specified, the type and ID are delimited by that character. If no delim$ is specified, then they are delimited by a tab (09).
getrangebycat$(seed$,first,count[,descending[,delim$]]) returns a linefeed delimited list of document types, IDs, and category indexes, in category index order, within the specified seed, which is provided as pipe-delimited segments. The list starts at the record number specified by first. Up to count records are returned. If descending is true (1), then the list is returned in descending order. If delim$ is specified, the type, ID, and category index are delimited by that character. If no delim$ is specified, then they are delimited by a tab (09).
getrangebydate$(date$,first|first$,count[,descending[,delim$]]) returns a linefeed delimited list of document types, IDs, and date/times, in date/time sequence, within the specified date, which is provided in yyyymmdd format. The list starts at the record number specified by first, or the time specified by first$ (in hhmmss format, 24 hour clock). Up to count records are returned. If descending is true (1), then the list is returned in descending order. If delim$ is specified, the type, ID, and date/time are delimited by that character. If no delim$ is specified, then they are delimited by a tab (09).
getrecentdates$(count) returns a linefeed delimited list of the most recent dates in the library. The count value specifieds the number of dates to return. Dates are returned in yyyymmdd format.
getsubdoc(doctype$,docid$,subid$,subdoc$) fills the subdoc$ property template with subdocument properties, and returns a 1 if successful. You can reference properties as subdoc.title$, subdoc.date$, subdoc.time$, subdoc.type$, and subdoc.size.
getsubids$(doctype$,docid$) returns a linefeed ($0A$) delimited list of sub IDs on file for the given document.
gettypes$() returns a linefeed delimited list of document types found in the library.
getyears$() returns a linefeed delimited list of years found in the library.
imgcount() returns the number of images (sub IDs) in the library.
movenext(doctype$,docid$[,docprop$]) returns information about the next document in sequence. Doctype$, docid$, and if supplied, the docprop$ document properties template, are filled. See the getdoc() method for information about fields in docprop$. The method returns 1 if successful, or 0 otherwise (such as an end of file error). The starting position of a series of move commands can be set with the moveto() method.
movenextcat(categories$,doctype$,docid$[,docprop$]) returns information about the next category and its associated document in sequence. Categories$, doctype$, docid$, and if supplied, the docprop$ document properties template, are filled. Categories$ is filled with a pipe-delimited list of category segments (note this is case-sensitive, returning segments as stored). See the getdoc() method for

information about fields in docprop$. The method returns 1 if successful, or 0 otherwise (such as an end of file error). The starting position of a series of category move commands can be set with the movetocat() method.

moveprev(doctype$,docid$[,docprop$]) returns information about the previous document in sequence. Doctype$, docid$, and if supplied, the docprop$ document properties template, are filled. See the getdoc() method for information about fields in docprop$. The method returns 1 if successful, or 0 otherwise (such as an end of file error). The starting position of a series of move commands can be set with the moveto() method.

moveprevcat(categories$,doctype$,docid$[,docprop$]) returns information about the previous category and its associated document in sequence. Categories$, doctype$, docid$, and if supplied, the docprop$ document properties template, are filled. Categories$ is filled with a pipe-delimited list of category segments (note this is case-sensitive, returning segments as stored). See the getdoc() method for information about fields in docprop$. The method returns 1 if successful, or 0 otherwise (such as an end of file error). The starting position of a series of category move commands can be set with the movetocat() method.

moveto(doctype$[,docid$) seeds the next movenext or moveprev methods to the position specified by doctype$ and docid$. To seed to the start of the library, provide a doctype$ of null. To seed to the end, provide a high doctype$, such as ff. Likewise, to seed to the end of a given document type, seed the docid$ with a high value like ff.

movetodate(ymd$[,hms$]) seeds the next movenext or moveprev to navigate the library in date/time order, beginning at the supplied date and optional time. The date must be in yyyymmdd format, such as "20091231". If the time is supplied, it must be in hhmmss format using a 24-hour clock, such as "150000" for 3:00 PM.

movetocat(seed$) seeds the next movenextcat or moveprevcat methods to an index position. Seed$ must be provided as a series of pipe-delimited category segments, such as "Vendors|000999|PurchaseOrders".

newdoc(doc$) fills an empty document property template as doc$. The property template contains the following fields:

- date$ (yyyymmdd format)
- time$ (hhmms format, 24-hour clock)
- entityid$
- notes$
- keywords$ (semicolon-delimited keyword list
- categories$ (semicolon-delimited list of pipe-delimited indexes)
- links$ (semicolon-delimited list of links)

newsubdoc(subdoc$) fills an empty subdoc property template as subdoc$. The property template contains the following fields: • title$ • date$ (yyyymmdd format) • time$ (hhmmss format, 24-hour clock) • type$ (extension of uploaded file) • size
putdoc(doctype$,docid$,doc$) updates the specified document properties. If the document does not exist, it will be added. Note that this method does not add any image files to the library. The doc$ template contains properties described in the getdoc method.
putsubdoc(doctype$,docid$,subid$,subdoc$[,filename$]) updates subdocument properties, and updates the image file as well, if filename$ is supplied. If the specified subdocument does not exist, it is added. The subdoc$ template contains properties as described in the getsubdoc method. The title, date, and time can be updated; other properties are ignored. Note that if you set the date and time values to null (""), they will be updated with the current date and time.
subdocexists(doctype$,docid$,subid$) returns true (1) if the document type, ID, and sub ID exist in the library, false (0) otherwise.

marked

a=new("marked"[,sessionid$])

The marked records object provides interaction with a browser sessions' marked records. A list of marked records is typically selected manually by the user while browsing document, and is maintained during the life of the user's browser session. The marked object allows adding to, deleting from, and navigating through the list of marked records.

A "marked" object is typically used in custom browser forms, when a user has marked one or more records and a related form rule set is designed to process that list. In addition, such a rule set can add or remove items from the marked record list. For example, it would be possible for a user to mark one record, then execute a form that would locate related documents and mark them as well.

Properties

sesid$ is a read-only value that returns the current session ID for the browser interface user.

Methods

add(library$,doctype$,docid$,subid$) adds a record to the marked records list. Returns 1 if successful, 0 if not (for example, if the record already exists in the list).
count() returns the number of records in the session's marked records list.
delete(library$,doctype$,docid$,subid$) removes the record from the session's marked records. Returns 1 if successful, 0 if not.
getmarked$() returns a list of all marked records as a string. The string is structured with linefeed ($0A$) delimited records, and tab ($09$) delimited fields. Each record consists of the library, doctype, doc ID, and sub ID fields.
movefirst(library$,doctype$,docid$,subid$) fills the four arguments with values from the first marked record. The list is sorted in library, doc type, doc ID, and sub ID order.
movelast(library$,doctype$,docid$,subid$) fills the four arguments with values from the last marked record.
movenext(library$,doctype$,docid$,subid$) fills the four arguments from the next marked record in sequence. Returns 1 if successful, 0 if not (such as at the end of the list).
moveprev(library$,doctype$,docid$,subid$) fills the four arguments from the previous marked record in sequence. Returns 1 if successful, 0 if not.
moveto(library$[,doctype$[,docid$[,subid$]]]) moves the record position based on the values provided. Subsequent movenext and moveprev methods will be relative to this location.

rac

a=new("rac")

Remote Access Control object for simplified document access from public users. The rac object allows creation and management of RAC codes, which are random codes linked to specific document images. RAC codes are very secure in that there are an extremely large number of possible codes (about 1.15e+77 combinations), so the potential for guessing a valid code and gaining access to a document is extremely small.

A remote access URL is in the form http://server/script?rac=*code*, where the code is a 44-byte encoded string previously generated for a specific document image. These URL's can be emailed to users, who can open the link and view the single file associated with that document image. Such a link does not require a login or password, so it useful in cases where a site doesn't wish to manage external user logins to provide full archive library access. A code expires after a certain number of days (default defined with racdays in uf80d.ini), as specified in the rac object's create$() method.

If the uf80d.ini [archive] external= value is set to a path to a public access cgi script (or direct to the internal web server, though that is not typically done), then the browser interface document email screen can generate or delete an RAC url. Note if this is not configured, rule sets can still generate a code and a url, by providing a script prefix to the geturl$() method.

Methods

count() returns the number of documents in the RAC table.
create$(lib$,doctype$,docid$,subid$[,validdays][,force]) generates code for the given document image, valid for the number of days (default - uf80d.ini). If a code has been previously generated, that code is returned and a new expiration date is calculated. To force the generation of a new code, use a force argument of 1.
deldoc(lib$,doctype$,docid$,subid$) removes the RAC code, if any, associated with the given document. This allows a rule set to explicitly revoke remote access for a given document.
getcode$(lib$,doctype$,docid$,subid$) returns the 44-byte code associated with the document image specified, or null if no code has been created.
getdoc(code$,lib$,doctype$,docid$,subid$) given a 44-byte RAC code, this fills the associated lib$, doctype$, docid$, and subid$ values. It returns a 1 if the document is found, or a 0 if not.
getexpire$(lib$,doctype$,docid$,subid$) returns the expiration of the RAC code associated with the given document. The format is yyyymmdd. If the document has no RAC code available, a null string is returned.
geturl$(lib$,doctype$,docid$,subid$[,scriptpath$]) returns full URL for a given document, which can be used to retrieve the document. If scriptpath$ is not supplied, then the 'external=' path is used from the uf80d.ini [archive] section. This script is the http: path used by a public user (not generally the internal http server) to access the archive server. Note this generally requires configuring a public-facing web server to use one of the CGI scripts supplied with UnForm. A script path example might be "http://mywebserver.com/cgi-bin/uf80a.cgi".

search

obj=new("search"[,searchid$])

The search object provides programmable control over the library search function offered by the browser interface and the command line -arcsearch option. By setting selected properties to values that follow the search syntax requirements, and running the search, a document list object is created that provides access to the results of the search. In addition, the results can be placed on a user's search results page, so a rule set can execute a search on behalf of a user, and the results can be displayed in the browser interface.

If a searchid$ is supplied, then the search object opens an existing search result, and the doclist object is attached to those results. This feature is useful when using custom browser forms driven from a search screen, as the search ID is provided and the search results the user generated can be accessed in a rule set. The search ID is the “sid” field value found in the URL of a search result screen, and is always a 10-digit value.

Each of the properties can be set to a wildcard (i.e. "**value*" or "*value**"), an exact value (i.e. "12345"), a range (i.e. "12/1/2007-12/31/2007"), or a regular expression with a tilde prefix (i.e. "~[0-9][A-Z]"). You can use "not" or "!" to look for archives that do not match a criteria, and "and"(or a semicolon) or "or" (or a comma), to search for multiple values or alternate values. All properties that are set must evaluate to true for a document to be added to the document list.

Searches are optimized when possible. The best optimizations are document IDs, entity IDs, small date ranges, and multi-level categories.

Note that document types and document IDs are case-sensitive when optimized.

Properties

docid$ specifies document ID(s) to search for. Note that document IDs are case-sensitive.
doclist is a document list object that is filled with search results when the run method is executed.
doctype$ specifies document type(s) to search. Note that document types are case-sensitive.
filter$ is compared to all document properties.
library$ contains one or more library names to search. Multiple libraries can be separated by semicolons.
recsread is a read-only property that stores the number of records read when the search was last run.
selected is a read-only property that stores the number of records selected and added to the document list object when the search was last run.
subids$ searches for documents that contain the subid(s) specified.
**text$** searches the contents of the @text subdocument, if found, for the text$ search criteria.
title$, date$, dateupdated$, entityid$, keywords$, notes$, categories$, and links$ specify criteria for a given document property. Dates can be specified in yyyymmdd[hhmmss] format, or in local delimited date format, such as "12/31/2009", where the mdy order is configured with datefmt=xxx in uf80d.ini.

Methods

makesearch(userid$[,description$[,errmsg$]) saves the search results in the user's search results table, so the browser interface will have access to the results. Optionally specify the description, or an error message, which will appear in the table.
run([append]) runs the search and fills the doclist object. If append is true (1), then the doclist object is not cleared before running. This allows multiple searches to produce a combined document list.

system

obj=new("system")

The system object provides access to operating system information and services. Note that the system object is a static object, meaning there is only one object instantiated for any and all new() functions executed.

Properties

All properties are read-only.

home$ is the path of the UnForm server's home directory.
networkid$ is the server's hostname or IP address.
osname$ is the specific operating system name, such as "MS-WINDOWS", or " UNIX-Linux-RedHat".
ostype$ is "win" on Windows, and "*nix" on Unix or Linux-like systems.
user$ is the user account running the UnForm server.

Methods

copyfile(from$,to$[,errmsg$]) copies the from$ file to the to$ file. If an error occurs, such as a permission violation or if the to$ file already exists, errmsg$ is filled with a message. The function returns true (1) on success, false (0) otherwise.
deletefile(filename$[,errmsg$]) erases the specified file. It returns true (1) if successful, or false (0) and fills errmsg$ if not.
getcmd$(cmd$[,output$]) executes the specified command, captures its standard output, and returns the contents of the standard output. If output$ is provided, it must be a file name that will also contain the standard output result.
renamefile(from$,to$[,errmsg$]) renames the from$ file to the name specified in to$. It returns true (1) if successful, or false (0) and fills errmsg$ if not.
runcmd(cmd$) executes the specified command and returns its exit code.
tempfile$([extension$]) creates a temporary work file that is erased at the end of the current job. The file name is returned. If extension$ is provided, the file will have that extension. Otherwise, a .tmp extension is used. The file must be opened to work with it. This function can be used to generate an empty file to use with other operating system commands.
tempfile([extension$]) creates and opens a temporary work file that is erased at the end of the current job. The channel number of the opened file is returned. If extension$ is provided, the file will have that extension. Otherwise, a .tmp extension is used. Use PTH(*chan*) to obtain the file name.
winprtprinters$() returns a linefeed ($0A$) delimited list of printer names available for *winprt* or *windev* printing on a Windows UnForm installation.
winprttrays$(printer$) returns a list of tray numbers and descriptions for the specified Windows printer. Windows print driver vendors often use user-defined tray numbers above 256, rather than tray numbers that match traditional PCL values. The list contains linefeed ($0A$) delimited tray lines, where each tray line contains a tab ($09$) delimited tray number and description.

textfile

obj=new("textfile"[,filename$])

The textfile class provides line-oriented text file processing capabilities. Each line in the file is considered a record, and lines are separated by linefeeds ($0A$) on Unix-like systems, and carriage return-linefeed ($0D0A$) sequences on Windows systems.

If filename$ is provided, the file's existing content is loaded, or it is created if necessary. If no filename$ is provided, a temporary file is created that is erased when the object is destroyed.

Properties

All properties are read-only.

filename$ contains the file name of the text file being managed.
lines contains the number of lines in the file.
found is an unkeyed collection object that holds the results of find operations. Each element of the collection is a matching line from the file.

Methods

append(row$) appends the line row$ to the end of the file.
delline(line) removes the line specified from the file, shifting remaining rows up.
find(search$[,nocase[,invert]]) initializes the found collection object, then fills it with text lines whose data contain the text search$. If nocase is true (1), then the search is case-insensitive. If invert is true (1), then records that do not contain search$ are added, rather than those that do. The function returns the number of records found.
findreg(regex$[,nocase[,invert]]) initializes the found collection object, then fills it with text lines whose data match the regular expression regex$. If nocase is true (1), then the search is case-insensitive. If invert is true (1), then records that do not match regex$ are added, rather than those that do. The function returns the number of records found.
findwhere(whereexpr$,dlm$,quotes) initializes the found collection, then searches records that match the where expression. For this search, records are assumed to be in delimited format, with the supplied dlm$ value as the delimiter. If quotes is true (1), then fields are parsed assuming they may be quoted to protect delimiter values in field values. The structure of the where expression is of a Boolean expression using fields, values, comparison operators, parentheses, and AND or OR, using #*number* syntax to represent field numbers. All fields are assumed to be string data, but you can use num() to convert strings to numbers, so long as the string is an unpunctuated numeric value. Here are some examples: #2<>"" - field 2 not null #2="100" and (num(#3)>=0 and num(#3)<10000) - field 2 is "100" and field 3 is between 0 and 10000

getline$(line) returns the line number specified.
insline(line,row$) inserts the line row$ into the file at the line specified. Lines are 1-based, so insline(1,"text line") inserts "text line" at the beginning of the file. If line is more than the number of lines in the file, new lines are added to ensure the file has at least that many lines.
purge() removes all lines from the file.
putline(line,row$) replaces the line specified with the contents in row$. If the line exceeds the number of lines in the file, lines are added as necessary.

webapi

obj=new("webapi")

The webapi object is used to generate URL strings appropriate for browser interaction with UnForm's document management web interface. These strings can be used when constructing responses to the UnForm Desktop Client.

Note: scriptpfx$ is only needed if server access differs from http access in use, for example if an external web server is used for document access even though the internal server is used for DTC, or a print-time rule set is executing the code.

Methods:

documenturl$(lib$,doctype$,docid$[,subid$,[nowrapper[,scriptpfx$]]]) returns a document URL. No subid will result in a document level view. A "@" subid will result in a conditional view, document if multiple sub ids are available, or @unform if just that is available. Otherwise, use a valid sub id. If nowrapper is 1, an image is shown without the standard UnForm document wrapper menus.
webform$(formname$,lib$[,doctype$[,docid$[,scriptofx$]]]) displays the indicated web form, with the library and optional document type and doc id populated on the web form.
catlist$(lib$,segments$[,scriptpfx$]) presents a category browse page, as if the user had clicked through category segments until a list of documents is displayed. Segments are pipe-delimited.
searchtpl(tpl$) returns search template tpl$ with the following fields: • library$ • doctype$ • docid$ • date$ • title$ • keywords$ • notes$ • categories$ • subids$ • text$ • links$ • dateupdated$ • entityid$ • filter$ Populate these fields as you would in a browser search, and pass this template to the searchurl$ method.
searchurl$(tpl$[,scriptpfx$]) returns search url with tpl$ based criteria. Note lots of criteria could exceed http GET size limits, which are typically a few hundred characters.

xmlreader

obj=new("xmlreader"[,filename$|content$])

The xmlreader class provides XML parsing capabilities to rule set code blocks. XML is a common file format for transmitting data between applications. It is a tag-based data format, where tags are used to identify data elements, as well as data attributes. Data is arranged in hierarchical fashion, where content contains other content, such as a report, which contains customers, which contain invoices, which contain shipping and order details, and so on.

The xmlreader navigates through this hierarchy using slash delimiters, similar to a file system directory, return data for a particular tag sequence. The data may be atomic data, a complete element, an element attribute, or it may be addition XML fragments that can be further parsed. For example, if the root element is "documents", and it contains a number of "document" elements, each of which contains an "invno" tag, you can navigate to an invoice number as "/documents/document/invno". If there are multiple occurrences of a particular element, you can get a count, and retrieve a particular element using a sequence number.

Elements with sequences can be referenced using a sequence number in many methods (referring to the last element tag), or can be referenced with [*n*] suffixes anywhere in the path chain. For example, "/documents/document[2]/invno" refers to the second document element's invno tag.

Element references can also be designed to match attributes or values, using one of two syntaxes:

- [*attribute*=*value*]
- [<*tag*>=*value*]

For example, "/documents/document[OrderID=12345]/Address[<Type>=ShipTo]/City" would locate the document with an attribute "OrderID" of "12345", then find the Address element of that document with a tag <Type> whose value was "ShipTo", then locate the Address element's City tag.

If a valid filename$ value is supplied, the file is loaded when the object is first instantiated. Alternatively, the argument can be an XML document string (if the value doesn't exist as a file, it is assumed to be XML content). At any time, a new file can be loaded or the XML content provided directly as a string.

Properties

body$ contains the non-header portion of the file, with any comments removed.
**content$** is the string representation of the XML data. When a file is loaded, content$ is filled with its data.
encoding$ contains the character encoding of the file. The default encoding is utf-8, which is a compressed version of unicode, but files may be encoded using other character sets, such as unicode or iso8859-1. Since UnForm's normal character set is iso8859-1 (or the 9J symbol set), it is often necessary to specify a target encoding when retrieving values for UnForm printing.
filename$ is the name of the most recently loaded file.
header$ contains the header portion of the file, similar to "<?xml version="1.0"?>".

root$ contains the name of the root element. All XML documents have a root element that is the parent of all other elements.
version$ contains the XML version number from the header.

Methods

getattr$(element$[,sequence],attrname$) returns the attribute value of the specified element. If a sequence is provided, the specified occurrence of the element is used.
getattrs$(element$[,sequence]) returns a list of element attributes, which are name-value pairs. Each pair is delimited by a linefeed ($0A$), and the name and value are delimited by a tab (09). If a sequence is provided, the specified occurrence of the element is used.
getchild$(element$[,sequence],child) returns a specified child element (which may itself contain nested child elements). If the sequence is provided, then it applies to the selection of the parent element, not the child element. The child argument is used to determine which child element to return. getchild$("/documents/document",5,1) returns the 1st child element of the 5th documents/document element.
getchildren$(element$[,sequence]) returns a linefeed ($0A$) delimited list of element names that are children of the element provided. If there are no children, then null is returned, and the value of the element is an atomic data value. If a sequence is provided, the specified occurrence of the element is used.
getchildren(element$[,sequence]) returns the number of child elements that are found under the element. If a sequence is provided, the specified occurrence of the element is used. The sequence applies to the element, not the children. getchildren("/documents/document",5) returns the number of child elements found under the 5th document element under the documents root element.
getcount(element$) returns the number of times a given element occurs. This can be used to determine how many sequences of a particular element are available.
getelement$(element$[,sequence]) returns element and enclosed content specified by element$. If a sequence is provided, the specified occurrence of the element is used.
getns$(element$[,sequence],uri$) scans the attributes of an element for an XML namespace declaration that matches the uri$ provided. It returns the namespace name, which can then be used to access elements that require a namespace prefix that associates a name with a URI.
getvalue$(element$[,sequence[,toencoding$]]) returns the inner value of the specified element. If the element holds atomic data, the result is a string data value. Note that an element may contain sub-elements. If a sequence is provided, the specified occurrence of the element is used. If toencoding$ is provided, then the value is translated from the XML document's encoding to the specified encoding. A common toencoding$ value might be "iso8859-1" or "9j" to translate UnForm's default 8-bit text encoding.
loadfile(filename$) sets the XML content by reading filename$
setcontent(content$) sets the XML content to content$.

Internal Variables

In addition to your own variables, UnForm provides a list of variables that you can use, or in many cases, set to a desired value.

across$	Can be set to values described in the across command. Available only in prepage and precopy.
bin$	Can be set to values described in the bin command. Available only in prepage and precopy.
cols$	Can be set to values described in the cols command. Available only in prepage and precopy.
copies **pcopies**	Can be set to the number of copies to generate for a page. You can change this value to dynamically adjust the number of copies. If the number you specify is higher than the number specified by the rule set, then that highest defined copy's text and enhancements will be repeated until your specified copies are complete. This value is reset after each page to the rule set default, so you can't set it in the prejob routine. If you set pcopies, that is also honored like the pcopies command.
copy	Contains the current copy number in precopy. Generally you shouldn't modify this value. If you need to skip printing of a copy, use the **skip** variable instead.
coverset$, coverfile$, coverargs$	If coverset$ is set to a rule set name$, a cover page is generated for the current document, using the rule set specified. Additionally, coverfile$ can specify a rule file, if the rule set is not in the current rule file, and coverargs$ can be set to command line arguments to use when running the subjob that generates the cover page. The cover page values must be set before the first page of the document is generated, so typically these values will be assigned in a prejob code block. However, if different cover page details are needed as the output device changes, you can set them in prepage or precopy code blocks.
crosshair$	Can be set to "Y" or "y" to enable crosshair grid printing over the output (laser and PDF output only).
down$	Can be set to values described in the down command. Available only in prepage and precopy.
driver$	Stores the current driver as "laser", "ps", "pdf", or "zebra". The win and winpvw drivers are considered variants of PDF, and driver$ is set to "pdf" when used. This variable should not be changed.
duplex$	Can be set to values described in the duplex command. Available only in prepage and precopy.
gs$	Can be set to the values described in the gs command. Available only in prepage and precopy.

lcopies$, ldarkness$, lspeed$	Set any of these values in a prepage code block to mimic the zcopies, zdarkness, or zspeed commands that can be used to control specific aspects of ZPL label printing.
margin$	Can be set to values described in the margin command. Available only in prepage and precopy.
noarchive	Set to 1 in prejob to turn off all archiving for the job, or in prepage to turn off archiving of just the current page.
nocover	Set to 1 to turn off cover page generation, before the first page of a document.
noemail	Set to 1 to turn off the execution of the email command. Note: this does not affect the email() code block function or direct calls to mailcall.
nohpgl	Set to 1 in prejob to turn off HP/GL formatting, as if the -nohpgl command line argument had been used. This variable is ignored in code blocks other than prejob.
nooverlay	Set to 1 to suppress an AFO overlay on a page, in either prepage or precopy.
orientation$	Can be set to "landscape", "portrait", "rlandscape", or "rportrait". It can also be set to a literal digit: "0"=portrait, "1"=landscape, "2"=reverse portrait, or "3"=reverse landscape.
outline$	Can be set to an outline string used when the PDF outline feature is turned on, by use of the outline command. Multiple levels of outlines can be defined by delimiting levels with vertical bars, such as outline$="Customer type "+get(1,6,4)+"|Page "+str(pagenum). This example would produce a 2-level outline structure with a customer type code being the top level, and page numbers as child levels.
output$	In laser output, this can be changed in prejob, prepage, or precopy, and is tracked by copy. Set it to the device or file name desired for output on the server. If it changes for a given copy in the middle of a laser job, UnForm will close the prior output channel and reopen the new one. This can be used to send a copy to a different printer, or to a fax device. You can set the value to any printer alias known to UnForm (in the unform.cnf file), any file, or a pipe or redirect, such as ">vfx -n "+faxnum$. When using a UNIX redirect or pipe, be sure to add quote characters (CHR(34)) around any data that might contain ampersands (&) or other shell-aware characters. For PDF output, you can set this value in the prejob code block to override any –o command line setting. Setting this value in any other code block is ignored.
pagenum	Can be referenced as the current page number. The value should not be changed.
paper$	Can be set to values described in the paper command. Available only in prepage and precopy.

rows$	Can be set to values described in the rows command. Available only in prepage and precopy.
showimages	Can be set to a non-zero value to execute an images command even if the skip variable is true. This is only honored in the precopy code block.
skip	Can be set to a non-zero value in prepage or precopy, to skip printing of that page or copy, respectively.
text$[all] **textpage$[all]** **textjob$[all]**	Stores the text for the page as a one-dimensional array. For example, text$[2] is the second line of text on the page. In prejob, it contains the content of the first page. In prepage and precopy, it contains the content of each page in sequence. You can use the array directly in code, or you can use the built in get(), mget(), set(), cut(), and mcut() functions to retrieve or manipulate its contents. Textpage$[all] contains the text of the current page before any manipulations by code blocks. Textjob$[all] contains the text of all pages in the job. Each line of each page, up through the last non-blank line of each page, is appended to the array when the input stream is initially parsed for the job. This array can be used in a prejob code block to analyze the full report content. PostScript input not supported.
tray$	Can be set to values described in the tray command. Available only in prepage and precopy.
uf.*xxx*$	A string template or composite string that can provide access to many attributes of the UnForm environment and command line.

uf.arcjob	Set to true (1) if the current job is a subjob for an archive command. This will be 0 otherwise.
uf.arcenabled	Set to true (1) if archiving is licensed.
uf.clientip$	The IP address of the connecting uf80c client. In the case of the Unix perl-based client, an attempt is made to identify the connecting terminal’s IP address, whether by telnet or ssh. The who command is used, which may provide a name from /etc/hosts rather than an IP address, but it will still be locally resolvable. In the case of jobs submitted via direct TCP/IP ports, the IP address of the computer that connected to the server’s listening raw port. Note that if this is a server-based printer share, the IP address returned will be of the server and not the originating computer.
uf.cols	Columns for the current page.
uf.copies	Copies defined for the job.

uf.deljob	Set to true (1) if the current job is a subjob for a delivery command. This will be 0 otherwise.
uf.dfrule$	Default rule file from the environment.
uf.driver$	Driver for the current job.
uf.emattach$	Command-line –emattach value.
uf.embcc$	Command-line –embcc value.
uf.emcc$	Command-line –emcc value.
uf.emfrom$	Command line –emfrom value.
uf.emlogin$	Command line –emlogin value.
uf.emmsgtxt$	Command line –emmsgtxt value.
uf.emoh$	Command line –emoh value.
uf.empswd$	Command line –empswd value.
uf.emsubject$	Command line –emsubject value.
uf.emto$	Command line –emto value.
uf.errfile$	Command line –e file value (dynamically determined by the server).
uf.home$	Home directory of the UnForm server.
uf.inputfile$	Command line –i file value (dynamically determined by the server).
uf.job	Current job number.
uf.jobexecerr$	If an error occurs while running a subjob with the jobexec() command, the error message will be in this variable.
uf.login$	Contains the login name from a -arclogin command option.
uf.maxdatacols	Maximum column with data on any page in the job.
uf.maxdatarows	Maximum row with data on any page in the job.
uf.maxpagecols	Maximum column with data in the current page.
uf.maxpagerows	Maximum row with data in the current page.
uf.maxpage	The maximum page number for the job. This value is calculated at the start of the job, and may be adjusted if pages are inserted or removed.
uf.model$	Command line –m model value.
uf.outputfile$	Command line –o file value. For server-based output, this is the –o option sent by the client. For client-based output, this

	is dynamically determined by the server.
uf.page	Number of input lines per page. Do not confuse this with the pagenum variable, which holds the current page number.
uf.paper$	Paper size name.
uf.parent	The job ID of the parent job when a subjob is running.
uf.pcopies	Pcopies defined for the job.
uf.pdfauthor$	Command line –pdfauthor value.
uf.pdfkeywords$	Command line –pdfkeywords value.
uf.pdfprotect$	Command line –pdfprotect value.
uf.pdfsubject$	Command line –pdfsubject value.
uf.pdftitle$	Command line –pdftitle value.
uf.prm$	Command line –prm value.
uf.rows	Rows for the current page.
uf.rulefile$	Command line –f rule file value.
uf.ruleset$	Selected rule set for the current job.
uf.shift	Horizontal shift value.
uf.subjob	Set to 1 (can be treated as a Boolean) if this is a sub-job executed by the jobexec() function.
uf.subst_file$	Command line –s file value.
uf.version$	The version of the UnForm server.
uf.vshift	Vertical shift value.
uf.warn$	Job warning messages, delimited by line-feeds. For example, to add your own message: uf.warn$=uf.warn$+"My message"+chr(10).

zcopies$, zdarkness$, zspeed$	Set any of these values in a prepage code block to mimic the zcopies, zdarkness, or zspeed commands that can be used to control specific aspects of ZPL printing.

Internal Functions

In addition to the intrinsic functions available in the run-time Business Basic engine, the most common of which are documented later in this chapter, UnForm provides a set of functions specific to its operating environment. Some functions are macros that perform an action, rather than return a value.

arrtostr(*arr$*[all],*str*$,*dlm*$)	Converts array arr$[all] to delimited string str$, using dlm$ as delimiter.
arrset(*arr$[all],col,row,cols,val$)*	Sets val$ into the one-dimensional array at the position specified. If val$ contains linefeeds, it is treated as a multi-row value, and the row increments for each additional row in val$. The array is redimensioned as necessary to accommodate the row and position.
basename(*file$)*	Returns the base name, without path information, from the file name specified. See also dirname().

bbxread(*file$*,*key$*,*rec$*,*errcode*)	Executes an instance of BBx, configured with the bbpath=*path* line in uf80d.ini, and obtains the record specified by key$ in file$. If an error occurs in the BBx instance, it is returned in errcode. An errcode value of –1 indicates no error occurred. The variable rec$ can be DIMed as a string template, but be sure to use '=10' to define field separators, as the default separator in the ProvideX engine is a hex 8A rather than the BBx default hex 0A. If it is not defined as a template, the raw record data is returned and may be parsed.
	
	Here is an example:
	
	`prepage{`
	`ky$=get(65,5,6)`
	`dim rec$:"id:c(5*=10), *:c(1*=10), …, fax:c(9*=10)"`
	`bbxread("/u/data/CUSTOMER",ky$,rec$,ec)`
	`if ec=-1 then faxnum$=rec.fax$`
	`}`

cdate(*datestr$* [,*fmt$*])	Converts a text date to a date (Julian) number, suitable for date compares, calculations, and use in the dte() function. The datestr$ can be a string in

	delimited format, based on the fmt$ string, or can be a simple string if digits in yyyymmdd[hhmmss] format. The format can be mdy, dmy, or ymd and a delimited date is parsed according to that order.
	
	The default date format is defined in uf80d.ini (datefmt=mdy).
	
	If a time is included, the value returned will include a fraction. Note that the dte() function requires an integer for the day portion, and the time portion, if provided, is an hours value:
	
	`x=cdate("12/31/2009 3:00 pm","mdy")`
	`x$=dte(int(x),fpt(x)*24:"%Mz/%Dz/%Yz %Hz:%Mz")`

clientenv(*name$*)	Returns a client-side environment variable value.

cmtocols(*centimeters*)
 	
cmtorows(*centimeters*)
	Returns columns or rows, given a centimeter measure.

cnum(*expression*)	Returns a number from a text string, after stripping formatting characters such as commas and dollar signs. Parentheses and minus signs indicate negative numbers. Use this function, rather than the intrinsic num() function, to convert text to numbers if the text can contain punctuation.
count(*str$*,*dlm$*)	Counts elements of a string, parsed by a delimiter.
countq(*str$*,*dlm$*)	Counts elements of a string, parsed by a delimiter, honoring quoted strings.
cstrans(*text$,fromcs$,tocs$*)	Returns text$ after translating it from one character set to another. Character set names include "uc", "utf16", or "utf-16" for unicode, or "utf8" or "utf-8" for UTF-8 encoding, or any of the character sets or symbol sets available in the UnForm unicode directory, such as 9j or 8859-1. The default character set is 9j, a pcl symbol set similar to ISO 8859-1.
cut(*col,row,cols,value$*)	Returns the value text at position *col*, *row*, for *cols* columns, after setting the specified position to *value$*. If *value$* is null ("") or spaces, cut()

	effectively erases the text. This is useful for moving data in text commands, such as **text 10,60,{cut(10,59,10,"")}**, which would cut text from 10,59 and move it to 10,60. PostScript input not supported.
dbconnect(*name$*[,*timeout*[,*errmsg$*]])	Connects to the database source identified by name$. The support server configuration is used to define the names and associate them with data source connection strings. Typically done in a prejob code block. Requires the Windows Support Server. If executed as a function, returns 1 on success, or 0 on failure.
dbexecute(*name$, command$, timeout, fdelim$, rdelim$, response$*[,*errmsg$*])	Executes the SQL command cmd$ and sets zero or more result rows in response$. Columns are delimited by fdelim$ (tab - chr(9) - by default). Rows are delimited by rdelim$ (CR-LF - chr(13)+chr(10) - by default). Requires the Windows Support Server, and a previously successful dbconnect() function execution in the current job. If executed as a function, returns 1 on success, or 0 on failure.
deliver(*filename$,to$,tags$* [,*response$*[,*errmsg$*]])	This function delivers the filename to a destination by fax or email, depending on the to$ value. The rule is simple: if to$ contains an "@" character, it is emailed. Otherwise, it is faxed. Tags are in the format name=value[,name=value ...]. Delimit each name=value pair with either commas or semicolons. The value can be quoted if it contains commas or semicolons. Tags are substituted with values in the fax/email configuration lines found in deliver.ini. The deliver.ini file contains configuration information for delivery methods. This file is documented in the Deliver Configuration chapter. See also the Deliver command, which simplifies subjob management for delivery while printing batches of documents.
delpage(*n*)	Command removes page n, pages n+1 to end are shifted down. PostScript input not supported.
dirname(*file$*)	Returns the directory portion of the file name provided. The value returned uses forward slashes

	on all platforms, and does not include a trailing slash. See also basename().
docidexists(*lib$*,*doctype$*,*docid$*)	Returns 1 if the document type and ID exists in the library, or 0 if not.
dtdel(*filename$*, *title$*, *userid$*, *ip$* [,*style* [,*errmsg*$]])	Deliver filename$ to the userid$ and/or IP address specified (the ip address can include a session suffix). The title$ will be presented to the user. A style of 0 indicates deliver for optional viewing, and 1 indicates immediate popup. If an error occurs, a message will be returned in errmsg$. See the Desktop Delivery and Forms chapter for details. If the filename$ parameter is a message starting with "msg:", it is treated as a popup message rather than a file. For example: "msg:Check the printer for invoices" would display a message window with the indicated text as the content.
dtform(*formname$*, *title$*, *userid$*, *ip$*, *datastr$*, *response* [,*timeout* [,*errmsg$*]])	Deliver the HTML form formname$ to the userid$ and/or IP address specified. The ip address can include a session suffix. The title$ value is presented to the user for approval. The datastr$ is a URL-encoded string that can be filled before the function is executed to provide default values for the form, and will contain URL-encoded data returned when the user submits the form. The response code will be 0 if the form was submitted, 1 if the user cancelled the form, 2 if the form timed out, or 3 if the user refused the form. The timeout can be specified to limit the amount of time the user has to accept the form (the default is 30 seconds). If an unexpected error occurs, a message is returned in errmsg$. See the Desktop Delivery and Forms chapter for details. If timeout is set to -1, then the user will not be prompted to accept the form. Instead, the form will be displayed immediately. However, if the user is not present or the monitor is not running, the form will not be responded to and no timeout notification will occur. Instead, the job will wait until terminated. This option should only be used if the user is guaranteed to be present, such as a follow up form to one that was previously accepted and submitted immediately before.

	The urlsetval and urlgetval functions can be used to create and parse the datastr$ values.
email(*to$, from$, subject$, body$, attach$, cc$, bcc$, otherheaders$, login$, password$,logfile$*)	Sends an email, assuming emailing is properly configured in the mailcall.ini file, using the information supplied. The arguments are positional but need not all be supplied. For example, **email(trim(get(81,1,40)), info@acme.com, "Please review", messagebody$)** will send a plain message to the address stored at column 81, row 1, for 40 characters in the current page. No attachment, carbon copy, etc. information will be used. As the arguments are positional, if you need to supply a login and password for the mail server to perform authentication, then all the arguments must be supplied, even if simply null (""). Note that this email function is different from the email command, in that the job itself is not sent, and multiple emails can be sent during the job stream within code blocks. This is useful, particularly in combination with the jobstore and jobexec functions, to develop batch email jobs.
entityencode(*str$*), entitydecode(*str$*)	Two functions that return str$ with HTML or XML entities encoded or decoded. For example, in HTML, the "<" and ">" characters are meaningful for character markup, so to reference those characters literally, rather than as markup, they are encoded to "&lt;" and "&gt;", respectively. When reading or writing data from a HTML page or XML document, it may be necessary to use these functions.
env(*name$*)	Returns the value of the operating system environment variable in *name$*, or in a literal quoted string. Returns null ("") if the variable does not exist.
err=next	May be used for any err=*label* option in any function or statement. Forces UnForm's error trapping to ignore an error. You may, of course, name your own err=*label* if desired.
exec(*expression*)	Executes a barcode, bold, box, erase, font, image, italic, light, micr, move, shade, text, or underline command from within the code block. *Expression* must be a single string value that contains the text

	of such a command, such as **exec("box "+str(col)+","+str(row)+",30,2.5")**. You can use the **exec**() function to add enhancements to a print job within the code block. The function can be used in either **prepage**{} or **precopy**{} blocks. Remember that some commands need quoted parameters to work properly. For example, if you exec() a text command, be sure to add quote characters around the text to be printed, using one of three methods: double any internal quotes, use an expression that uses 22 for quotes, or use an expression that uses CHR(34) for quotes. For example, **exec("text 10,10," + chr(34) + message$ + chr(34) + ",cgtimes,10")**, or **exec("text " + str(col) + "," + str(row) + ",""Quoted Text"",univers,12")**.
exists(*file$*)	Returns 1 (true) if the file path specified exists, 0 (false) otherwise.
fileext(*file$*)	Returns the extension of file name provided, without the leading ".". The extension is the segment of the name after the last ".".
finddata(*text$[all], search$, coloffset, rowoffset, columns, result$[all]*)	Searches the text array for a search string, and returns each item found in the result$ array. The data returned is based on the locations found, offset by the coloffset and rowoffset values, and the length specified. Each result is trimmed of leading and trailing spaces. The function returns the number of positions found, and the result$ array is indexed from 0 to the number of positions minus 1. Result$[0] is the first item found, result$[1] is the second, and so on. This function is useful for report mining, where specific types of rows contain the data desired. See the findpos() function for a description of how the search$ string is interpreted.
findpos(*text$[all],search$,result[all]*)	Searches the text array for a search string, and returns the number of locations found. Each location found is returned in the response[all] array. The result array is structured as follows: result[0,0]=first column result[0,1]=first row

	result[1,0]=second column result[1,1]=second row The first dimension contains each position found, from 0 through the number of positions minus 1. The second dimension contains column numbers in element 0, and row numbers in element 1. The search string can be a simple string, or a tilde (~) followed by a regular expression. If the string contains a "@" character, then two, three, or four comma-separated digits should follow, indicating the starting column,row, and ending column,row to search in the text array. If the ending row is not supplied, search ends at the last row in the array. If the ending column is not supplied, all columns from the first column are searched. "Total@50,45,54,66" will search for the word "Total" in columns 50 through 54, rows 45 through 66 in the array.
fromuc(*text$,charset$*)	Converts unicode text to single-byte text in the character set specified. Characters that are not present in the character set are replaced with a "?". Returns the single-byte text. Known character set tables are specified in the UnForm unicode directory. Also, "utf8" or "utf-8" can be used to convert UTF-16 (Unicode) format to UTF-8, a common encoding for XML or HTML data.
get(*col,row,cols)*	Returns text from the text$[all] array, without substring or array out-of-bounds errors.
get(*col,row,cols,trim$*)	Same as get(), but with a trim "Y" or "N" option.
get(*col,row,cols,trim$,page*)	Same as get(), but with a trim "Y" or "N" option, and a page number option to retrieve information from any page of the job.
get(*col,row,cols,trim*)	Same as get(), but with a Boolean trim (0 or non-0) option.
get(*col,row,cols,trim,page*)	Same as get(), but with a Boolean trim (0 or non-0) option and a page number option.
getaddress(*book$,entityid$,doctype$,address$*)	Opens the address book specified, and fills the address$ template with data from the address book entry for entityid$ and doctype$. The address book template can be referenced as address.entityid$, address.doctype$, address.entityname$,

	address.contactname$, address.sendto$, and address.combine. The function returns 1 if successful, or 0 if not.
getarc(*lib$,doctype$,docid$,subid$, filename$* [,*errmsg$*])	Retrieve an archive image to a user-specified or temporary file.
getcolumn(*rows$*,*column*[,*first*[,*count*[,*fdelim$*[,*rdelim$*]]]])	Slices a column from a block of delimited rows, returns fields delimited by $0A$. fdelim$ defaults to 09, rdelim$ to $0A$. If first and count are supplied, slice begins at row "first" and continues for "count" rows. This provides easy pagination capabilities. If count=0, slice continues to last row.
getdocidprop(*lib$, doctype$,docid$, prop$*)	Sets prop$ to a composite string containing properties about the document specified. These properties include: Prop.date$ - date in yyyymmdd format Prop.time$ - time in hhmmss format (24-hour clock) Prop.title$ - title string Prop.entityid$ - entity id string Prop.notes$ - notes, which can have CRLF line breaks Prop.keywords$ - semi-colon delimited keywords Prop.categories$ - semi-colon delimited categories with pipe-delimited segments Prop.links$ - semi-colon delimited list of links If the document type and ID does not exist in the library, each of the fields in the composite string will be empty. Use the docidexists() function to determine if a document exists.
getfile(*filename$*)	Returns contents of filename$ as a string. Can be used to load a file into a string.

getfilefield(*filename$,key$,field*)
 	
getfilefield(*filename$,key$,field, dlm$, quoted*)
 	
getfilerec(*filename$,key$*)
	Returns a record or field from a text file, given a key that matches the first field in each record. The dlm$ field is a field delimiter, such as "," or chr(9) for comma or tab delimiters, and the quoted field is a Boolean (0=false, non-0=true) that indicates fields may be quoted, as would be the case in a classic csv file. If no matching key is provided, the functions return an empty string. If no dlm$ and quoted

getfilerec(*filename$,key$, dlm$, quoted*)	parameter is supplied, then a classic comma-separated-value format is presumed (dlm$=”,”, quoted=1). These functions provide an efficient way of providing data to UnForm from applications. For example, an application could export customer IDs and email addresses, and UnForm could lookup addresses by customer ID. Files are parsed once and cached until they change, so subsequent retrievals are very fast. Caching is permanent (across jobs). Keys are limited to 127 bytes, so the first column must be limited accordingly. In a quoted file, fields that contain a quote character must escape that character with a backslash, like "Board - 1' 2\" length".
getinival(filename$,section$[,name$])	Returns the section or, if name$ is supplied, the value of the name in the section specified, of the .ini formatted file specified. .ini files are organized in to sections via [*name*] headers, and lines within the section contain name=value pairs. When a full section is returned, each line is delimited by a linefeed character (chr(10) or $0a$). This can be useful in cases where data is stored in .ini file format and UnForm needs to access it. If filename$ doesn't exist, it is treated as ini file content.
getpage(*n,arr$*[all])	Fills text array arr$[all] with page n data lines. PostScript input not supported.
getpaircount(*values$* [,*delim$*])	Returns the number of delimited pairs in value$. For the string "id=00100,name=Acme Corp", the count would be two. The default delimiter is a comma.
getpairvalue(*values$*,*number[,delim$*])	Returns the value of the specific name=value pair in the values$ string. Pairs are delimited by commas unless delim$ is specified. The first pair is number 1, the second is number 2. getpairvalue("id=00100,name=Acme Corp",2) returns Acme Corp.

getpairvalue(*values$*,*name$ [,delim$*] [,*casesensitive*])	Returns the value associated with the name from the delimited list of pairs in values$. The default delimiter is a comma. If casesensitive is true (1), then the name must match exactly. If the name is not found in values$, then null is returned. getpairvalue("id=00100,name=Acme Corp","name") returns Acme Corp.
getpatternvalue(*pattern1$*,*pattern2$*, *array$[ ]* [,*erasepat*,[*includepat*]])	Searches each element of the one-dimensional array for text that starts with pattern1$ and ends with pattern2$. If either is null, returns from the beginning or end of the line. If either starts with ~, balance is a regular expression (use \~ to enable ~ as a search character). The text data found is returned, each element separated by a linefeed ($0A$), optionally with the pattern(s) included if includepat is true (non-zero). If erasepat is true (non-zero) then the pattern and text found is removed from the line on which it is found. Only the first match in each array element is returned.
getpatternvalue(*pattern1$*,*pattern2$*, *searchtext$* [,*erasepat*,[*includepat*]])	Searches searchtext$ for text that starts with pattern1$ and ends with pattern2$. If either is null, returns from the beginning or end of the line. If either starts with ~, balance is a regular expression (use \~ to enable ~ as a search character). The text data found is returned, optionally with the pattern(s) included if includepat is true (non-zero). If erasepat is true (non-zero) then the pattern and text found is removed from the line on which it is found. Note searchtext$ must be passed as a variable, rather than a literal or expression, for this to work. Only the first match in searchtext$ is returned.
getppdval(*name$*,*option$*)	Returns a value from the PPD file associated with the job, either a default file selected by the –p driver command line option, or one explicitly named with a –m command line option. PPD files are generally

	used by PostScript printers to define command sequences for settings like duplex, bin, and tray selection. The laser driver can also use a custom PPD file for defining PCL sequences for various printer options. This function can be used to retrieve control sequences for use in boj, eoj, bop, or eop values.
getsubids(*lib$,doctype$,docid$*[,*dlm$*])	Returns a list of document archive sub ID's, delimited by linefeeds or by the specified delimiter.
gettrans()	Returns the active translation file.
getuserprop(*userid$,prop$*)	This function fills the template variable prop$ with attributes for the specified user. It returns 1 on success, or 0 if there is an error. If there is an error, the template contains empty fields. The template fields can be referenced as: Prop.username$ Prop.email$ Prop.entityid$ Prop.companyname$ Prop.telephone$ Prop.groups$ (semicolon-delimited groups the user is a member of)
gproperty(*name$*)	Returns the DSC structured comment value of the given name from a AFO print stream. The most common use for this is to obtain a document title from the PostScript data, such as title$=gproperty("Title"). Comments can be found in the header and trailer of the PostScript data.
gtextcount(*page*)	Returns the number of text elements found on the specified page. This function is valid only for AFO jobs.
gtextitem(*page,item,text$*[,*col* [,*row* [,*cols* [,*rows*]]]])	Fills text$ with the specified text element identified by page and item, where item can range from 1 to the number of text elements on the page. If supplied, will fill col, row, cols, and rows variables with the position and size of the text item. This function is valid only for AFO jobs.
gtextfind(*page, pattern$, txt$[all], rects[all]*)	Scans the specified page for text elements matching the specified pattern. For each element selected, the text is added to txt$[*n*], and the position and size of the text is added to rects[*n*,0-3], where 0 is

	the column, 1 is the row, 2 is the columns, and 3 is the rows. Pattern$ can be a simple text phrase, which must be contained in a text element to be selected, or it may be in the format "~*regexp*" to search for a regular expression. In addition, it may have a *@col1,row1,col2,row2* suffix to limit the search to text elements enclosed by the specified rectangle. To search for a value that includes a literal "@", use "\@". The function returns the number of elements selected, which can be used to as the range of valid elements placed in txt$[all] and rects[all]. If the function returns 2, then txt$[1] and txt$[2] contain the first and second text elements selected. This function is valid only for AFO jobs.

imgx(*imagefile$,units*)
 	
imgy(*imagefile$,units*)
	These functions to return image x and y dimensions, helpful when trying to scale an image to actual size. Units indicate what is returned: 0=pixels, 1=inches, 2=cols/rows.
	
	The function supports jpg, bmp, tif, and png formats. If the file can't be opened or parsed, 0 is returned.

inchtocols(*inches*)
 	
inchtorows(*inches*)
	Return columns or rows, given a measurement in inches.

inspage(*n,arr$*[all])	Inserts text array arr$[all] as page n, shifting existing pages as necessary. If n is any number greater than the highest page number, or -1, a page is appended (i.e. inspage(999,x$[all]) will add page 3 to a 2-page job. PostScript input not supported.
jobclose(id$…)	Closes and erases the temporary storage file associated with id$. Open jobs are all automatically closed at the end of the primary job.
jobids(*dlm$*)	Returns a list of job ID values that are active, separated by the delimiter character specified. Any ID supplied in a previous jobstore() command and not closed with a jobclose() command will be

	returned.
jobexec(*id$,output$,driver$,argstring$ [,async]*)	Executes a sub-UnForm job using the parameters given. The id$ identifies a job with one or more pages previously stored with the jobstore() function. The output$ value defines where the sub-job's output should go. This can be a file name, like "/archive/"+invoice$+".pdf", a device name, like "//printsrv/hp4000", or a pipe/redirect, like ">lp –dhp4000 –oraw". The driver$ argument can be set to one of the –p drivers supported by UnForm, such as laser or PDF. The argstring$ contains any additional command line parameters you wish to add to the sub-job command line. You can use any parameter supported by the uf80c client, though the -i, -o, and -p options are specified using the other three function arguments. A rule set can check uf.subjob, as "if uf.subjob" or "if uf.subjob=1", to test if an instance is running from a jobexec() function. The optional *async* flag can be set to a non-zero value to force the job to be executed asynchronously, so the jobexec() command returns as soon as the subjob is queued. This operates via the rpq directory, which means the subjob will execute within a few seconds, as long as there is a job license available. If not, the job will remain queued until a license is free. Note that if you use this flag, the main job must be able to operate without the subjob output. For example, if the main job is designed to email or fax a result of a subjob, it will fail. Such processing should be moved into the subjob's execution context.
jobfile(*id$*)	Returns the temporary text file associated with id$.
jobstore(*id$* [,*array$[all]*])	Stores the content of the current page in a temporary file, identified by id$. The value in id$ is user-defined, and each unique value stores content in a different temporary file. The other job-related functions use the id$ value to select which file to use. For example, you could store a whole job with an id$ of "job", and individual documents in jobs identified by their document number. Each would be stored separately and could be jobexec'd

	separately. If the optional array is supplied, then it, rather than the current page content, is written to the work file. If supplied, the array must be a one-dimensional string array (i.e. dim myText$[66]).
lbound(*arr$*[all][,*dimension*])	Returns the lower-bound of the array arr$[all]. If arr$ contains multiple dimensions, you can specify which dimension. For example, if arr$ is dimmed as x$[100,1:2], lbound(x$[all])=0, lbound(x$[all],2)=1.
left(*str$,length*)	Returns the leftmost *length* characters from str$, padding with spaces on the right to enforce *length*. Note also the mid() and right() functions.
libexists(*lib$*)	Returns 0 if library lib$ doesn't exist, or 1 if it does.
log(*msg$*)	Writes a log entry to the server log file, usually uf80d.log.
log(*msg$,logfile$*[,*format$*])	Logs a message (time stamped) to the specified file. The file is created if necessary. If *logfile$* is null, the server log is used. If *format$* is supplied, it is used as the mask for the time stamp. The following character sequences are substituted in the format: • %Yl – 4 digit year • %Yz – 2-digit year • %Mz – 2 digit month • %Ms – short month name • %Dz – 2-digit day • %Ds – short day name • %Hz – 2-digit hour (24 hour clock) • %hz – 2-digit hour (12 hour clock) • %mz – 2-digit minute • %sz – 2-digit minute If no format is supplied, this format “%Yl-%Mz-%Dz %Hz:%mz:%sz”. If the format is supplied but is null (“”), then no time stamp is written.
logwarn(*msg$*)	Adds a message to the job's .err file, which is also presented when the design tool runs a preview that results in warning errors.
lower(*expression*)	Returns text in lowercase.

ltrim(*str$*)	Returns the value of str$, trimmed of leading spaces.
mcut(*col,row,cols,rows,value$,lf$,trim$*)	Returns multiple lines of text, optionally with line-feed delimiters and/or trimmed of spaces. The lf$ argument can be set to "Y" or "y" to add a line-feed character between each line; likewise, the trim$ argument can be set to "Y" or "y" to cause each line to be trimmed before returned. In addition, mcut() assigns each line in the cut region to *value$*. Use null ("") or spaces to erase the source text. PostScript input not supported.
mget(*col,row,cols,rows,lf$,trim$*)	Returns multiple lines of text into a single string, optionally with a line-feed delimiter and/or trimmed of spaces. This function is useful in conjunction with multi-line functionality of the text command. The lf$ argument can be set to "Y" or "y" to add a line-feed character between each line; likewise, the trim$ argument can be set to "Y" or "y" to cause each line to be trimmed before returned.
mid(*arg1$,arg2,arg3*)	Safely returns a substring without generating an error 47 if the value in *arg1$* isn't long enough to accommodate position *arg2* and length *arg3*. Note also the left() and right() functions.
mset(*col,row,cols,rows,value$*)	Multi-line set function. Will work with multi-line value$, delimited with mnemonic \n character sequences or chr(10) values. PostScript input not supported.
msfax(*filename$, faxnum$, tags$ [, errmsg$]*)	Faxes filename$, normally an UnForm-generated PDF file, to the fax number specified in faxnum$. Numerous supported tags can be specified in tags$, in the format tag1=value,tag2=value,... Requires the Windows Support Server. For more details, see the Windows Support Server chapter.
parse(*str$,n,delimiter$*)	Returns the *n*th element of the string str$, when parsed by the delimiter specified. For example, parse("one,two",2,",") would return "two". If the delimiter is null, then any white space delimiter is used.
parseq(*str$,n,delimiter$*)	This is the same as parse(), except that honors quoted values in the string str$, ignoring delimiters contained in them.

pdfpages(*pdffile$*)	Returns the number of pages in a PDF file. The file must be in non-optimized format.
pdftoimage(*fromfile$,tofile$,format$*[,*resolution*[,*errmsg$*]])	Uses Ghostscript, local to the server or via the Windows Support Server, to convert from PDF file fromfile$ to an image file tofile$, using the format format$. Valid formats match those of the Ghostscript drivers defined in uf80d.ini.
prm("*name*")	The prm() function has been added as a synonym to the gbl() and stbl() functions, which return global string table values typically associated with the –prm command line option.
proper(*expression*)	Returns text in Proper Case.
putaddress(*book$,entityid$,doctype$,address$*)	Opens the address book specified, and adds or updates the address entry identified by entityid$ and doctype$. The address$ template can be created by execution of a getaddress() function, even from an invalid entityid$, then filled with appropriate values. The entityid$ and doctype$ arguments are automatically copied to the address$ template. The address book template fields can be referenced as address.entityid$, address.doctype$, address.entityname$, address.contactname$, address.to$, and address.combine.
putdocidprop(*lib$, doctype$, docid$, prop$*)	Updates the document properties of the document type and ID in the library specified. The document properties are replaced with the values found in the composite string prop$. These string properties are: Prop.date$ - date in yyyymmdd format Prop.time$ - time in hhmmss format (24-hour clock) Prop.title$ - title string Prop.entityid$ - entity id string Prop.notes$ - notes, which can have CRLF line breaks Prop.keywords$ - semi-colon delimited keywords Prop.categories$ - semi-colon delimited categories with pipe-delimited segments Prop.links$ - semi-colon delimited list of links All properties found in the string are updated, so you must first read existing properties using the getdocidprop() function, then modify those

	properties desired, then update them with this function. This function will not add new documents to a library. It only updates existing ones.
putpage(*n,arr$*[all])	Replaces page n with text array arr$[all]. PostScript input not supported.
right(*str$,length*)	Returns the rightmost *length* characters from str$, padding with spaces on the left to enforce *length*. Note also the left() and mid() functions.
rtrim(*str$*)	Returns the value of str$, trimmed of trailing spaces.
sdocmd(*object$, cmd$, response$, errmsg$*)	sdOffice® is a product from the publisher of UnForm that provides Microsoft Office® automation capabilities to network computers. When used in conjunction with UnForm, sdOffice can become part of a report mining platform and can be used to augment UnForm jobs with Microsoft Office functionality. This function sends a command to the sdOffice object specified. If the command returns data (all get* commands return data), the response will be returned in response$. If an error occurs, a message will be returned in errmsg$, and the function will return False (0). If no error occurs, the function returns True (1). If the sdOffice communication settings have not been initialized before the first execution of this function, a default environment is created. This environment uses uf.clientip$ as the target sdOffice office client, localhost port 6114 as the default sdOffice server, and 30 seconds as the default timeout.
sdoinit(*target$, timeout, server$, port*)	Sets the sdOffice communication environment. The target$ value is the machine running the sdOffice Office client, which interprets commands sent via the sdocmd() function, and defaults to uf.clientip$. It is not uncommon, however, for sdOffice Office Clients to be configured to service multiple users, so this value might need to be specified. The timeout value defaults to 30 seconds, and is

	used to limit the amount of time that the sdocmd() functions will wait for a response to a command. The server$ and port arguments specify the machine where the sdOffice server runs, and the port it listens on for application connections. These default to localhost and 6114, respectively.
set(*col,row,cols,value$)*	Returns *value$*, after it places *value$* in the text$[all] array at the position indicated. PostScript input not supported.
setlogin(*userid$,password$*)	Sets the login and password to enable the library object to access a library. An administrator can define secure passwords in the browser interface, and a password ID can be used in place of a plain text password. The syntax is simply the ID with a "store:" prefix in the password$ field, such as "store:SQLUser1".
settrans(*filename$*)	Sets the translation file dynamically as the job runs. This overrides what might be set via a -trans command line option.
sqlconnect(*datasource$*[,*user$* ,*pswd$* [,*otheroptions$* [,*errmsg$*]]])	Connects to the data source specified, using the specified user, password and optional parameters. Returns a channel on success, or 0 on failure, in which case the errmsg$ argument contains an error message. An administrator can define secure passwords in the browser interface, and a password ID can be used in place of a plain text password. The syntax is simply the ID with a "store:" prefix in the pswd$ field, such as "store:SQLUser1". The channel returned is used for subsequent access to the database using sqlexecute() and sqlfetch() functions. See the Database Access chapter for more information about database support.
sqlexecute(*chan*,*command$*[,*errmsg$* [,*result$* [,*fdelim$* [,*rdelim$*]]]])	Executes the SQL command specified, typically a SELECT statement, on the channel specified. The channel must have been previously returned by a sqlconnect() function. If result$ is supplied as an

	argument, then a sqlfetch() method is executed to fill result$ with all rows, and the number of rows fetched is returned.
sqlfetch(*chan*,*result$*[,*count* [,*errmsg$* [,*fdelim$* [,*rdelim$*]]]])	Fills result$ with some number of rows from the most recent SQL command executed on the channel with the sqlexecute() function. Returns the number of rows filled. The number of rows returned is determined by the count argument. If not supplied, 1 row is returned, allowing a loop to be processed one row at a time. If count is -1, then all available rows are returned. The default field delimiter is a tab (09), but this can be specified with the fdelim$ argument. The default row delimiter is a linefeed ($0A$), but this can be specified with the rdelim$ argument. If an error occurs, errmsg$ will hold a message. If no more rows are available, the function returns 0 and result$ is empty.
sshost(*server$,port*)	Sets the Windows Support Server hostname and port. Default values are defined in the uf80d.ini file in the sshost and ssport settings. This command allows for dynamic changing to a different server.
striplines(*text$*)	Returns text$, stripped blank lines from multi-line text, such as addresses. As a byproduct, all CR characters are also removed, leaving simple LF line delimiters.
strtoarr(*str$,arr$*[all],*dlm$*)	Converts string str$ to an array arr$[all], by splitting str$ on delimiter dlm$
sub(*str$,old$,new$*)	Returns a string where all occurrences of old$ in str$ are replaced with new$.
subidexists(*lib$*,*doctype$*,*docid$*,*subid$*)	Returns 1 if the document type, document ID, and sub ID exists in the library, or 0 if not.
tempfile([*ext$*])	Creates and returns the name of a temporary file that is removed automatically at the end of the job. The file will have a ".tmp" extension, unless an extension argument is provided.
textimage(*text$, font$, size, cols, rows, color$, charset$, errmsg$*)	Creates a bmp image using Image Magick based on the TrueType font, point size, and image size. For limited Unicode output, you can use this function instead of font embedding to avoid large output

	caused by embedding a full ttf font. If a server-based version of Magick is unavailable, a Windows Support Server can be configured to support this function. The font$ value is either a name found in the [fonts] section of the ufparam.txt file, or an actual TrueType font file name. The size specifies the point size for the text. The cols and rows define the image size. If cols is 0, then the image will be sized to exactly fit the text. Otherwise, the text is centered within the image. The color setting can be "rgb rrggbb" (rr, gg, and bb are hexadecimal values 00-FF), or any color that Image Magick recognizes. The charset defaults to Windows ANSI (9J or iso8859-1). Text is converted to utf-8 for Magick. The function returns the name of the temporary .bmp file, which can be used in an image command expression. Direct unicode can be supplied in text$ if charset$ is "uc". To ensure the image isn't re-scaled incorrectly, use imgx/imgy functions to obtain actual cols/rows sizes for use in the image command.
textfile(*path$*)	This function creates a file and returns a path name to that file. The value of *path$* is interpreted in three ways. If null (""), a new temporary file is created, and will be erased automatically when the job is complete. Optionally, the value may start with a period to force the extension of the temporary file to match the value, such as ".pdf". Otherwise, the value should be a full path, and that file will be created and returned. Such custom paths are not erased automatically at the end of the job.
textheight(*text$, fontnum*\|*fontname$, size, attr, cols* [*,linespacing*])	Returns the text height, in rows, of text$, given the font number or name, size in points (or pitch if the font is mono-spaced),style attribute (0=normal, 1=bold, 2=italic, 3=bold italic), columns (for wrapping calculations), and optional line spacing. This will be the equivalent height that would be

	used by the text command given the same attributes. If linespacing is not supplied, the default is based on the current lpi.
textwidth(*text$, fontnum*\|*fontname$, size, attr*)	Returns the text width in columns of text$ given the font number or name, size in points, and style attribute (0=normal, 1=bold, 2=italic, 3=bold italic). The function honors the same font mapping as is used in regular UnForm processing for pdf, and understands the same fonts that are understood for internal calculations for justification, where laser fonts are loaded from the standard fonts.txt file, pdf fonts are mapped from these, and postscript fonts are loaded from .afm files in the psfonts directory. For Postscript, the width is based on the Windows ANSI symbol set. If the font is a TrueType font, and the text value doesn't appear to be unicode text, UnForm will implicitly convert it to unicode. If the text is preencoded to unicode but does not contain any null (00) characters, add 0000 to the string to prevent this implicit conversion.
touc(*text$*,*charset$*)	Converts single-byte text in the character set specified to unicode text. Returns the unicode text. Known character set tables are specified in the UnForm unicode directory. Charset$ can also be "utf8" or "utf-8" to convert from UTF-8 to UTF-16 (or Unicode).
translate(*name$* [,*context$*], *forcecontext*)	Returns the value associated with the specified name, based on the translation file and current rule set. The context value can be "text", "barcode", or "anchor", and if *forcecontext* is true (non-zero), only context-based names are searched.
trim(*expression*)	Returns *expression* after trimming spaces from the left and right side.
ttfchars(*fontnum*)	Returns a string made up of Unicode characters, two bytes per character, found in the True Type font mapped from the font number provided.
ubound(*arr$*[all][,*dimension*])	Returns the upper-bound of the array arr$[all]. If x$ contains multiple dimensions, you can specify which dimension. For example, if arr$ is dimmed as x$[100,1:2], ubound(x$[all])=100, ubound(x$[all],2)=2.

upper(*expression*)	Returns text in UPPERCASE.
urlgetfld(datastr$,name$)	Returns the value of the name$ field. The value is returned without URL encoding. mailto$=urlgetfld(datastring$,"to")
urlsetfld(datastr$,name$,value$)	Returns a URL-encoded string with the field name$ set to value$. The field is added if necessary. datastring$=urlsetfld(datastring$,"to",someone@somewhere.com)
urldelflds(datastr$,names$)	Returns the a URL-encoded string after removing the fields specified in name$ from the URL-encoded string datastr$. Multiple fields can be separated by commas. datastring$=urldelflds(datastring$,"to,from,subject,body")

urlgetnames$(datastr$)	Returns a list of field names in the data string.
	
	fldlist$=urlgetnames$(datastring$)
	count=parsec(fldlist$,",')

When using variables and line labels, you should avoid using any values that begin with "UF". UnForm reserves all such variables and labels for its use. You may use a backslash (\) at the end of a line to continue the statement on the next line. Lines prefixed with "#" are not added to the code.

Two data elements from the command line can be referenced in code blocks using the stbl() function (use gbl() in ProvideX environments). The –s *sub-file* option will generate stbl values as "@*name*". For example, if the substitution file contains the line 'company=Smith Produce', then stbl("@company") will return "Smith Produce". Further, the –prm command line option will directly create stbl values.

Runtime verbs and functions

The following list is a summary of verbs and functions present in the UnForm runtime engine and are commonly used in UnForm applications. Note that all functions accept an ",err=*linelabel*" or "err=next" argument, and all verbs accept the same after any parameters, to branch if an error occurs. Optional arguments are shown inside braces {}.

ASC(string)	Returns the ASCII numeric value (0-255) of the first character of *string*.
ATH(string)	Returns a binary equivalent of a human readable hex string. ATH("1B") returns an escape character.
BIN(integer,length)	Returns a binary integer representation of the specified length. The inverse function of this is the DEC() function.
BREAK	Breaks out of a loop structure. Equivalent to EXITTO *linelabel* if *linelabel* is the line after the closing WEND or NEXT.
CHR(integer)	Returns a character string whose ASCII value is *integer*, between 0 and 255. CHR(27) returns an escape character.
CONTINUE	Executes the next iteration of a loop structure. Equivalent to GOTO *linelabel*, if *linelabel* is the closing WEND or NEXT.
CVS(string,arg)	Returns a converted string according to the cumulative value of the integer *arg*. Values: 1=strip leading spaces, 2=strip trailing spaces, 4=uppercase, 8=lowercase, 16=non-printable characters to spaces, 32=multiple spaces to single spaces. CVS(a$,3) trims both leading and trailing spaces.

DATE(julian {,time} {:mask})
 	
DTE(julian {,time} {:mask})
	Returns a human readable date and/or time, based on the Julian date (see the JUL() function), a decimal time (hour and fraction of hour – 12.5=12:30PM), and a format mask. The mask can contain combinations of placeholder characters and modifiers. The placeholders are %M=month, %D=day, %Y=year, %H=hour (24 hour clock), %h=hour (12 hour clock), %m=minute, %s=second, %p=AM/PM. Modifiers include z=zero fill, s=short text, l=long text. Examples on June 30, 1999 at 1:30 in the afternoon: date(0) returns "06/30/99", date(0:"%Ml %D, %Yl") returns "June 30, 1999", date(0,tim:"%hz:mz %p") returns "01:30 PM".

DEC(string)	Returns the decimal conversion of the binary integer in *string*. The counterpart to the BIN() function. To treat *string* as an unsigned integer, you should use the form DEC(00+*string*).

DIM string(length {,char})	Returns a string of *length* size, of spaces or the specified *char* character.
DIM name[dim1{,dim2{,dim3}}]	Creates a numeric or string array variable. Dimensions can be simple integers, indicating an index range of 0..*dim*, or two integers separated by a colon, like 1:12.
DIR("")	Returns the current disk directory. On Windows, DIR(*driveletter*) will return the current directory for the specified disk drive.
EPT(number)	Returns the 10's exponent value of *number*. EPT(100)=3, EPT(12)=2.
ERASE filename	Erases a file. Obviously, care should be taken to only erase temporary work files.
EXITTO linelabel	Exits a loop structure (current level only, in nested structures) and jumps to the specified *linelabel*.

FBIN(number)
 	
I3E(number)
	Returns a 64-bit IEEE number in natural left to right ordering.

FDEC(string)
 	
I3E(string)
	Returns the decimal value of a 64-bit IEEE number.

FID(channel)	Returns a file identification string for the file opened on *channel*. For devices, just the device name is returned. For files, the first byte indicates the file type (00=indexed, 01=serial, 02=keyed, 03=text, 04=program, 05=directory, 06=mkeyed, etc.) You can verify a file is a plain text file like this: test$=fid(filechan); if test$(1,1)=03 then x$="text file".

FILL(integer{,string})
 	
DIM(integer{,string})
	Returns a string if *integer* length, made up of successive iterations of *string*, or spaces if no *string* is provided. FILL(7,"abc") will return "abcabca".

FIN(channel)	Returns additional file information not found in the FID() function. A common use of this function is to determine file size, which is stored as a binary integer in the first four bytes. To get the length of a file: x$=fid(filechannel); length=dec(00+x$(1,4)). Additional potentially useful information can be found as well. See the language reference manual for more details.
FOR numvar=start TO end {STEP increment}	Initiates a loop, using a numeric variable initialized to *start* the first pass through the loop, incrementing by 1 or the specified *increment*, which can be negative, until the variable exceeds (or goes below in the case of a negative *increment*) *end*. The statements following this command,

	until a NEXT *numvar* statement, are executed. The loop can be broken from with the BREAK or EXITTO verbs.
FPT(number)	Returns the fractional portion of a number. FPT(100.66) returns .66.
GOSUB linelabel	Jumps to the specified *linelabel*. Statements will be executed until a RETURN verb is encountered, and execution will return to the statement after the GOSUB.
GOTO linelabel	Jumps to the specified *linelabel*.
HTA(hexstring)	Returns a human readable hex string of *hexstring*. HTA(CHR(2)) returns "02". HTA("0") returns "30".
IF test THEN statement(s) {ELSE statement(s)} {END_IF or FI}	Conditionally executes *statements*. *test* must be a simple expression that produces a Boolean or numeric result (0=false, non-0=true). Multiple statements can follow the THEN or ELSE clause by separating them with semi-colons. Statements following a END_IF are executed without regard to the condition of the last IF test. Nested IF statements are accepted without practical limit.
INT(number)	Returns the integer portion of a number. INT(99.645)=99.
JUL(year,month,day)	Returns the Julian integer of the specified date elements. The year should be specified, if possible, as a 4-digit year. Otherwise the function will assume a century of 1900. The complement of this function is the DATE() function.
LEN(string)	Returns the length of the string.
LET var=value{,var=value…}	Assigns variables to values. The variables can be numeric, string, or array variables. The values can be any compatible numeric or string expression. LET is implied when an assignment is performed in context. "LET a=1" and "a=1" are equivalent.

MASK(string{,regexpr})
 	
MSK(string{,regexpr})
	Returns the position where a regular expression pattern was found in the *string*, or 0 If not found. If *regexpr* is not specified, then the last *regexpr* used is re-used. This provides a performance benefit for repeated uses of the same *regexpr*. The length of the string matched is returned by the TCB(16) function.

MAX(num{,num…})	Returns the largest number found in the list of *num*s.
MIN(num{,num…})	Returns the smallest number found in the list of *num*s.
MOD(num1,num2)	Returns the remainder of dividing *num1* by *num2*. MOD(4,3)=1, MOD(6,3)=0.

NUM(string)	Returns the decimal value of a string, assuming the string is a well-formatted value containing digits, a single optional period (decimal point), and a single optional leading hyphen (minus sign). Other punctuation or characters will return an error. NUM("-12.5") returns 12.5. NUM("1,456") results in an error.
ON integer GOTO|GOSUB linelabel{,linelabel…}	Branches to one of the indicated line labels based on the value of *integer*. If *integer* is 0 or less, branch to the first label, 1 to the second, 2 to the third, and so on. The last label is used for *integer* values greater than that of the last label.
OPEN(integer{,err=linelabel|next}{,isz=integer}) string	Opens the file named in *string* on channel *integer*. To open a file in binary mode regardless of the file type, specify a block size with the ",isz=*integer*" option.
POS(string1 relation string2 {,increment {,occurrence}})	Scans *string2* for a substring having the specified *relation* to *string1*. POS("B"="ABC") returns 2. POS("B"<"ABC") returns 3. The string can be searched in even character increments: POS("02"="002002",2) will return 5, since the second and third characters, though matching the search string, are not located at an increment boundary. If the string is not found, or the requested relation, increment, and occurrence cause the string to not be found, the function returns 0.
PRINT(channel) value {,value…}{,}	Prints a series of values, numeric and/or string, to the file channel specified. A line-feed character is added to the channel unless the last character of the statement is a comma.
READ{ RECORD}(channel {,options}) variable {,variable…}	Reads data from the specified channel into the specified variables, looking for field terminator characters to delimit variables. Field terminators include line-feeds, carriage returns, and nulls. Valid *options* include "err=*linelabel*", "end=*linelabel*", "siz=*blocksize*". "key=*keystring*", "ind=*index*", and "dom=*linelabel*". For intrinsic keyed files, use the key= or ind= options to read specific records. For text files, use READ to process line-feed delimited files, but be aware that carriage return characters act as field separators. To read text files as binary files, use READ RECORD with a "siz=" option.
REM	Places a non-executing remark line in the code. In UnForm, you can also use a # character.
RETRY	Retries the statement that caused the last error branch to be taken.
RETURN	Returns from a GOSUB branch.

RND(integer)	Returns a pseudo-random number. The random number sequence can be re-seeded by providing a negative integer, so it is common at startup (like in a prejob code block) to seed the RND function with a variable number, such as MOD(JUL(0,0,0)+INT(TIM*10000),32000). The *integer* can be a number from –32767 to +32767. Positive numbers return a random integer from 0 to *integer*-1. If *integer* is 0, a random number between 0 and 1 is returned.
ROUND(number,precision)	Returns *number*, rounded to *precision*. ROUND(1.566,2)=1.57. ROUND(100.83,0) returns 101.

SCALL(string)
 	
SYS(string)
	Executes the operating system command in *string*. Returns the result code provided by the operating system. Use this function to interface with the operating system or external commands. This is an alternative to opening a pipe to a command.

SETERR linelabel	Provides a generic error handler to catch errors not trapped by err=*linelabel* branches in functions and verbs. UnForm also adds error handling code to code blocks, and reports errors in a job error file (temp/*jobno*.err in the server directory).
SGN(number)	Returns a 1, 0, or –1, depending on the sign of *number*.

STBL(string1{,string2})
 	
GBL(string1{,string2})
	Returns and/or sets the global string table value named *string1*. If *string2* is present, then the string table is set to *string2*. In both cases, the value is returned. If *string1* has not been set, STBL(*string1*) will result in an error (trappable with err=*linelabel*, of course).

STR(number{:mask})
 	
STR(string{:mask})
	Converts a number to a string, optionally formatted with a *mask*. The mask can contain any text, plus the following placeholder characters: 0=zero filled digit, #=space filled digit, "."=decimal point, ","=thousands separator, -, (,), and CR for negative numbers. STR(99.91:"0000.00") returns "0099.91". STR(19093.255:"###,##0.00") returns "19,093.26".

STRING filename{,err=label}
 	
SERIAL filename{,err=label}
	Creates a text file of the name specified. Use either a string variable or expression, or a quoted literal string.
	
	Examples: STRING "/tmp/test.txt" or STRING "/tmp/"+str(dec(info(3,0)))+".txt",err=next.

TCB(integer)	Returns task control information. Commonly used *integer* values include: 10=last operating system error code, 16=length of MSK() function match, 20 for number of arguments passed to an ENTER command.
TIM	Numeric variable that returns the decimal time of day, from 0.0 to 23.99.

UNT	Numeric variable that returns the next available file channel number.
WHILE condition…WEND	Looping construct that performs statements between WHILE and WEND statements as long as *condition* is true or non-zero.
WRITE {RECORD} (chan,options)data	Writes data to a file. Numerous options are available, some depending on the type of file. See the full programming documentation available on www.pvxplus.com for more details.

Lexical Substitutions

With the change in Version 6 to the ProvideX run-time engine, it is possible that some BBx syntax in code blocks will be incompatible. For the most part, the lexical substitutions automatically performed by UnForm will handle any differences, with the exception of direct I/O to BBx data files, which can be handled with the bbxread() function. However, if any additional substitutions are required, they can be entered into a user-defined text file called uflexsub.usr.

The format for the lines in this file is simply *bbxsyntax=pvxsyntax*. An example is provided in uflexsub.txt, which is a file that provides some standard syntax substitutions that the internal lex capabilities do not support. You can add your own by simply creating uflexsub.usr and adding lines.

Error Codes

When code is executed, any errors that are not handled by err=*label* branches are reported as warnings on a job trailer page. High error code numbers are used to report errors in client-server communication. Common error codes are shown in the following table.

Error Number	**Description**
1	End of record error, which may occur on a buffered disk write operation if the data is too long for the record buffer. This error is rare in UnForm jobs, but could occur if output is being printed to a printer alias defined in the config.unf file.
2	End of file, which may indicate a disk full message, or a file that is too large for the operating system to handle.
10	An invalid file name was given.
11	A missing key on a keyed read operation, or a duplicate key on a keyed write operation with a DOM= option.
12	A missing file error on a file open operation, or a duplicate file error on a file creation operation.
13	Normally a file permission error.
14	A file channel conflict or locking conflict error.
16	Out of resources, such as file handles. If this error occurs, it is often due to opening too many files. This can easily occur if files are opened but not closed in a loop or call construct.
18	Normally a file or directory permission error.
20	Syntax error. Common causes include mismatched parentheses, incorrect spelling of verbs or functions, or missing or incorrect function arguments.
21 or 25	Missing statement, as referenced in an ERR=*label*, or a goto or gosub branch.
23	Missing GBL/STBL variable name, or missing string template variable.
26	String/Number mismatch, where a string variable or literal is used where a number is expected, or visa versa.
27	Stack error, such as a return without a gosub, or a wend without a while.
28	For/Next error, such as executing a next without an associated for.
29	Mnemonic error. Mnemonics are pre-defined codes inside single quotes, such as 'FF' or 'LF'. Therefore, single quotes are not valid as string literal indicators; only double quotes are.
30	Corrupt program, which indicates that UnForm itself is probably corrupted, unless this error occurs on a call statement referencing an external program.
31	Out of memory.
33	Out of memory.
36	Mismatched arguments on a call statement.
40	Numeric overflow, normally caused by a divide by zero.
41	An integer overflow or range error. Some functions require integer arguments, so a floating point number will cause this error. Also, some functions require integer arguments to fall in a certain range, and this error will occur if the function is given a value outside of the valid range.

Error Number	**Description**
42	Array subscript error.
43	Masking error.
46	String length error.
47	Substring error, such as a starting position of 0 or a length greater than the length of the string.
997	The client's IP address is not in the server's list of valid addresses. To correct this problem, the allow= line in the server's uf80d.ini file must be modified to match the network addresses in use, and the uf80d server restarted.
998	The maximum number of concurrent jobs licensed was exceeded.
999	The server was unable to start the secondary process to handle the job within the allotted time of 30 seconds. Possible causes include a sluggish server and network problems, such as a DNS server timeout.
1024	The Windows uf80c.exe client can report this error if the network connection to the server is too slow.
1057	The Windows uf80c.exe client can report this error if the server is not running or a firewall is blocking the primary listening port.

EMAIL INTEGRATION

UnForm includes a copy of the MailCall utility that enables emailing of attachments from within UnForm. This is most often used to send PDF files. It can be used to email laser printer (PCL5) files, as long as you know the email recipient has a compatible printer that supports any of the fonts used in your documents. If you use CGTimes, Courier, and Univers fonts, then any PCL5 laser print device should be able to properly print documents, as long as the user can copy the file directly to the printer.

The MailCall utility is used internally by the deliver command, the email command, which emails a complete PDF-formatted job, and the email() function, which can send email(s) in mid-job, possibly with attachments resulting from sub-jobs managed by the job*xxx* series of code block functions. These two features are capable of handling most email requirements. However, within a code block, you can use the MailCall program directly, for any degree of control required. For example, the MailCall utility provides logging facilities that are helpful in debugging connection or communication problems. To implement logging, direct calls to the MailCall program are required.

Generally, the only requirement to get email working is to configure the server= line in the mailcall.ini file. This line needs to name the machine or IP address of the SMTP server that MailCall connects to. Other configuration options serve as default values.

The remainder of this chapter discusses the utility in depth.

Configuration

To configure MailCall, you need to edit the mailcall.ini file, using any text editor. If you don't have a mailcall.ini file, then you can rename mailcall.sds to be mailcall.ini. The following notes provide details about each option.

The most important element of the configuration is to ensure the system that executes MailCall has connectivity to your SMTP mail server. This may be an in-house system, or it may be hosted by your Internet Service Provider. A fairly foolproof way to test this is to telnet to port 25 on the mail server from your system (telnet *hostname* 25 from either UNIX or an MS-DOS Command Window). If you get a non-error response, MailCall should work.

server=*smtp-server*
This contains a reference to the IP address or domain name of the SMTP email server. This is used by the native socket interface, the mailcall.exe program, and the mailcall.pl program. If your mailer= setting uses sendmail or mmdf, this value is not used.

port=*port-number*
When native sockets are used, the default SMTP port of 25 can be overridden by setting a *port-number*. Normally, this should not be required.

from=*email-address*
Defines a default 'from' address if none is supplied when sending email.

hostname=*hostname*
If the environment does not provide a system name that is valid for the SMTP server, you can specify a value here. If no value is specified, then MailCall will determine the system hostname with the UNIX "hostname" command, or on Windows with the INFO() function in Visual PRO/5 or the NID variable in ProvideX. This element is only used by the native socket support.

login=*username*
password=*password*
If the SMTP server requires authentication, then you can define a default *username* and *password* with these elements. It is also possible to specify a *username* and *password* within the CALL interface. These values, if required, are supplied by the mail administrator, and must be supplied exactly as specified or you will probably get an authentication error and be unable to send mail.

mailer=*commandline*

NOTE: **When running MailCall under UnForm, there is no need to configure a mailer= line.**

If MailCall will *not* use internal sockets, then this line configures how MailCall actually sends the mail. If you are running under ProvideX or PRO/5 or Visual PRO/5 revision 2.2 or higher with a proper alias line defined, MailCall will use internal sockets and this line does not need to be configured. When required, BBx executes this command line via the SCALL() function. There must be a % character in the command line, which MailCall substitutes with the email submission file at run-time.

If no mailer value is set (all lines are commented) and a mailer is required, then a default mailer line is constructed, using "perl mailcall.pl % >mailcall.pl.log 2>mailcall.pl.err" on UNIX or "mailcall.exe %" on Windows. The proper path to the mailer is automatically generated. In other words, **if you have Perl or are on Windows, there is generally no need to configure a mailer= line**.

On Windows, *commandline* should be set to the full path for mailcall.exe plus the % argument, such as 'c:\mailcall\mailcall.exe %'. Be sure to use DOS-style backslashes rather than forward slashes.

On UNIX, you will probably want to use mailcall.pl. mailcall.pl should be in the same directory as the MailCall program, and mailer should be set to the full path to mailcall.pl. The *commandline* should be 'perl /usr/mailcall/mailcall.pl % >/dev/null' (adjust the directory path as necessary). Perl, of course, must be installed on your system for this to work. To enable logging, change the ">/dev/null" to ">*pathname*", and the conversation that mailcall.pl has with the SMTP server will be logged to that file.

If you use sendmail, the *commandline* '/usr/lib/sendmail –t <%' should work.

If you use mmdf, then the *commandline* 'echo $LOGNAME >%2; cat % >>%2; /usr/mmdf/bin/submit -uxto,cc* <%2; rm %2' is used to submit email messages. The command line argument "-uxto,cc*" instructs submit to scan for To: and Cc: headers for addresses.

Note that mmdf doesn't support Bcc: headers, while the other three methods do.

timezone=*zone*
Internet mail must include a date and time header; a properly formatted time will include your time zone. On Windows, the *zone* is added to the date and time header in the submission file. On UNIX, the time zone is determined from the date command.

charset=*charsetname*
The default character set in Internet email is "us-ascii". With this setting, it is possible to override this default for text elements of an email that includes attachments, including the body text itself.

Most configuration options have equivalent variables in the CALL string template. If you define values in the template, they override the equivalent values in the configuration file.

Implementation

Implementing MailCall requires the use of code blocks to establish temporary output files and then the execution of MailCall itself.

Here is a sample PDF rule file that can be used to email a PDF document. Since the pdf driver can only be used to produce one PDF file at a time, there is only one file to worry about.

```
[mailpdf]
cols 80
rows 66

prejob{
# set output file to a unique name using process ID
# note the pdf driver only allows output changes in prejob
output$="/tmp/email"+str(dec(info(3,0)))+".pdf"
}

postdevice{
call uf.home$+"mailcall.pv",1,x$,""
x.to$="someone@somwhere.com"
x.subject$="PDF Report attached"
x.msgtxt$="Here is a sample PDF file.\n"
x.attach$=output$
x.from$="sdsi@synergetic-data.com"
call uf.home$+"mailcall.pv",0,x$,""
erase output$
}
```

Here is a slightly more complex example, designed to email the second copy of a PCL document. PCL allows output to be split in the middle of the job, so this technique would work in a batch run where a document reference number is used to define the output name. This sample assumes the report will contain the email address at column 1, row 1 of each document.

```
[mailpcl]
cols 80
rows 66
copies 2

prejob{
# initialize mailer$ template
call uf.home$+"mailcall.pv",1,mailer$,""
}

precopy{
# set copy 2 output to document number plus extension
if copy=2 then output$=get(70,6,6)+".pcl"
}

postdevice{
# whenever the document number changes, this routine is executed
if copy<>2 then goto skip_mail
mailer.to$=trim(get(1,1,40))
mailer.subject$="Report attached"
mailer.msgtxt$="Here is the report you asked for.  Copy it to your laser printer.\n"
mailer.attach$=output$
mailer.from$="sdsi@synergetic-data.com"
call uf.home$+"mailcall.pv",0,x$,""
erase output$
}
```

MailCall Reference

```
CALL uf.home$+"mailcall.pv", mode, dat$, errmsg$
```

You may call either mailcall.pv or mailcall.bb; both are identical files for use within UnForm.

Arguments:

mode is an integer value that controls how MailCall interprets or returns data in the dat$ argument. The following are valid mode values:

0	Send mail based on data in string template dat$
1	Return a string template suitable for mode=0 in dat$
2	Return version information in dat$

For modes 0 and 1, **dat$** is a string template in the format:

from:c(1*=0),to:c(1*=0),cc:c(1*=0),subject:c(1*=0),otherhead:c(1*,msgtxt:c(1*=0),attach:c(1*=0),status:n(1*=0),forcebase64:n(1*=0),forcenotify:n(1*=0),bcc:c(1*=0),bodymime:c(1*=0),charset:c(1*=0),timeout:n(1*=0),statuspause:n(1*=0),dialog:n(1*=0),login:c(1*=0),password:c(1*=0),logfile:c(1*=0),timezone:c(1*=0),charinterface:n(1*=0),logdata:n(1*=0)"

To provide for additions to this base template, you should always use a single CALL using mode=1, which will return a usable template in dat$.

For mode 2, dat$ returns a printable string that describes the version and license status.

Here is a description of each template field:

dat.from$ contains the sender's email address. This value defaults to what is specified in the "from=*address*" line in mailcall.ini

**dat.to$** contains one or more email addresses delimited by commas. Note that if multiple addresses are desired, it is more common to place additional addresses in the cc$ field. Each address should be structured in one of two ways: *name@domain* or "*text name*" <*name@domain*>. It is important that if any data is present other than the plain internet email address, that the Internet address be enclosed in angled brackets <>.

dat.cc$** contains zero or more carbon copy addresses. Multiple addresses must be delimited with commas. Address formats are the same as for **dat.to$, above.

dat.bcc$ contains zero or more blind carbon copy addresses. Multiple addresses must be delimited with commas. A blind carbon copy address receives a copy of the email, but the Bcc: header is removed from the submission, so no other recipients know of the Bcc: recipients.

dat.subject$ contains a single line of subject text, describing the message content.

dat.otherhead$ contains additional mail headers, should they be necessary. The rfc822 specification allows for user defined headers starting with the characters "X-", in the format of "X-*name*: *value*". Each header line should be suffixed with a CRLF (or LF) delimiter ($0D0A$). There must be no blank lines in this value, and all lines should have a proper header structure of 'name <colon (:)> <space> value'.

dat.msgtxt$ is plain text for the message body. It may contain line breaks delimited with CRLF (or LF) sequences. Lines should not exceed 900 characters without line breaks. You may also use UNIX-style line break escapes (\n sequences) instead of binary CRLF characters.

**dat.bodymime$** can be used to define an alternate body text (dat.msgtxt$) MIME type. The default is "text/plain", but it is common to prepare message body text as HTML, in which case you can specify dat.bodymime$="text/html". This must be a well-known standard value (see the mime.typ file included with MailCall), and should be of the text/* family.

dat.attach$ contains one or more file names to attach to the message, delimited with commas. If this contains names, then MailCall will produce a MIME-encoded message, with the message body as plain text, text-style files (MIME types such as text/plain or text/html) as quoted-printable attachments, and other files as base64-encoded attachments.

dat.status, if set to 1 (or any positive value), will cause a status window to display as the email is processed. This flag is honored when MailCall uses native sockets or the external mailcall.exe program. When native sockets are used, the status window operates for both generation and SMTP server submission. When the external Windows mailer is used, it only operates for submission. External UNIX mailers do not support this flag.

For logging on UNIX installations, if you are using mailcall.pl, do this:

- Verify the setting of $log=1 in mailcall.pl near the top of the program
- Direct stdout to a file or the screen by modifying the mailer= line: something like "perl /usr/mailcall/mailcall.pl % >/tmp/mailcall.log". or just "perl /usr/mailcall/mailcall.pl %".

dat.statuspause can be set to the number of seconds to pause before closing the status window after the SMTP conversation is complete. This can help the user see the process completion without a quickly flashing window. This flag is only honored when MailCall uses native sockets and the dat.status flag is set.

dat.forcebase64, if set to 1 (or any non-zero value), will cause MailCall to always encode files with base64-encoding. By default, files whose MIME type is text are encoded using quoted-printable encoding.

dat.bodymime$, if set, will override the default text/plain MIME type used for the message body.

dat.charset$, if set, will override the charset default defined in the mailcall.ini configuration file, or the default of "us-ascii", when no setting is defined. Character sets are associated with any text body or attachment.

dat.login$, dat.password$, if set, and if the SMTP server requires authentication, are used for the AUTH LOGIN authentication process. These values would be provided by the ISP or mail server administrator, and must be provided exactly as specified. These values are honored when MailCall uses native sockets or the mailcall.exe or mailcall.pl mailers.

dat.logfile$, if set to a pathname, will trigger detail logging of the SMTP conversation when MailCall is using native sockets. The file will be erased and created each time MailCall is CALLed. Be careful not to use pathnames that should not be erased.

dat.timezone$, if set, will override the normal time zone value that is applied to the Date: header. The default time zone comes from either the timezone= value in mailcall.ini (for Windows) or the UNIX 'date +%Z' command. Use this to set a relative GMT value, like "-0800" for PST.

dat.charinterface, if set to a non-zero value, will force character-mode for the dialog and status window displays, even in a GUI environment. The status window display affected is only the internal version used when native sockets are utilized, not the status window displayed by the mailcall.exe mailer.

dat.logdata, if set to a non-zero value, and if the dat.logfile$ is defined, and if a native socket is in use, will cause the mail submission file data to be logged to the log file specified in dat.logfile$. The default behavior is to only log SMTP conversation information and suppress the message data.

errmsg$ will contain the text of an error message, if one occurs.

UnForm Notes: When UnForm is running on a UNIX system, there is no usable terminal device associated with it, even if run from the command line. Therefore, the user interface options (such as dat.dialog=1) of MailCall are not available. This is not the case on a Windows installation, so long as the server is running as an application rather than a service. Note however, that any user interface presented occurs where the UnForm server is running, not necessarily where the client runs.

HTML OUTPUT

UnForm provides an optional capability to produce HTML files from reports, using a processing engine that is similar to that used for laser printer output. Using this capability, users can convert their standard text-based reports into HTML documents, which are suitable for viewing with Web browsers such as Netscape Navigator and Communicator, and Microsoft Internet Explorer.

Reports can be converted in real-time, as part of a CGI or ASP procedure that responds to a browser request to generate a report, then format it as HTML. Alternatively, reports can be converted with a periodic batch process, such as a nightly procedure that produces various reports, then converts them all to HTML for viewing the next day.

Even without a rule set, UnForm can streamline text reports by producing plain text pages with horizontal rules at the end of each page. These are constructed using HTML templates, so standard company headers and footers can be applied even to reports that are not enhanced via a rule set.

Creating HTML

UnForm will create HTML output if you specify "-p html" on the command line. Given this parameter, and with no "-f *rulefile*" parameter, UnForm will look for the "html.rul" file rather than the default "unform.rul" file used for printer output.

By default, the HTML output is generated to standard output (on UNIX only), but it is normally preferable to specify an output file, such as "-o /usr/internet/docs/reports/aging". UnForm can then build the reports with varying styles in stages, and a browser can view interim results as soon as the first page is generated. UnForm will add a ".htm" extension automatically to the output file. UnForm will also create additional files depending on the style of the report. For example, if a table of contents is generated as a separate document, then the base file (aging.htm in the above example) will be the table of contents, and additional files will be generated for the pages of the report (aging.*page*.htm).

A sample command, therefore, might look like this:

unform -i aging.txt -o /usr/internet/docs/reports/aging -p html -f ourhtml.rul

As HTML structure is very different from that of laser printers PCL, HTML rule sets are very different from printer rule sets. UnForm uses HTML table structures to format pages. These structures have a defined hierarchy of rows, cells, and data, with attributes applied to either cells or data. HTML rule sets follow this structure in that you define rows, then within rows you define cells, and then within cells you define the attributes of the cell and text.

The HTML output that UnForm produces can be in one of several styles. The rule set options used to trigger the style are shown in parentheses:

- The simplest form is that of one document with all the pages sequentially created as tables. If no output file is specified (-o *filename*), this is what UnForm will produce regardless of any style options you specify.
- The output can be produced in one file, with a table of contents at the top of the file (toc=y or toc=l, multipage=n). As each page is generated and appended to the file, the table of contents is updated and inserted at the top. The table of contents consists of descriptions linked to the individual pages. The descriptions default to "Page number *n*", but can be created in page code blocks. Additionally, the table of contents can be created as a vertical column (toc=y), or as a bullet list (toc=l).
- The output can be produced in multiple files (multipage=y), with the table of contents being the primary one, with links to each page as a separate HTML document.
- The output can be produced as frames (frame=y), with the table of contents in one frame, and pages in the other. The target pages can be stored in a single file, multi-page document, or with each page in an individual file.

Note that all these options but the first require that a table of contents be maintained as each page is generated. In order to construct an updated document as each page is generated, UnForm must generate temporary files with which to build the HTML required. The *filename* specified by the "-o" option is re-created as each page is completed. Therefore, if standard output is generated rather than output files, only the first style can be produced.

This interim generation of files means that the HTML output can be viewed as soon as the first page is generated. This can be very helpful when large reports are being formatted in real-time.

HTML Configuration

When generating HTML documents, UnForm uses several configuration elements to structure the output. Most of these are created in UnForm's parameter file, which is named "ufparam.txt". Note that you can create a custom parameter file for your site that will not be overwritten during an update of UnForm by copying "ufparam.txt" to "ufparam.txc". Then make any changes to the custom version.

A section in the configuration file headed by "[html]" controls HTML configuration. It will look like this:

```
[html]
page=page.htm
toc=toc.htm
both=both.htm
frame=frame.htm
pagenum=Page number
imagelib=
imageurl=
complete=Report Complete
incomplete=Report not complete (reload page to view again)
```

The following table describes each parameter:

Element	**Description**

page=*filename*
toc=*filename*
both=*filename*
frame=*filename*
	These elements point to HTML template files in UnForm's home directory. These files are used by UnForm based on the style of output being generated.
	
	To create custom templates for your site, you should copy each file to some other name, modify the file names identified in these four elements, and edit the templates for your needs.
	
	See "HTML OUTPUT TEMPLATES", below, for more information.

colwidth=*text*	The default column cell width is *text*. This can be a pixel value, such as "colwidth=9", or any other value accepted by a <td width=*value*> tag in HTML. If no value is specified, UnForm uses "2em", which indicates 2 *half-characters*, based on the average width of a character in the default font. This value can also be specified for individual reports using the **colwidth** keyword in a rule set.
pagenum=*text*	This text is used to generate the default table of contents' values. A space and the page number follow the text.

Element	**Description**
imagelib=*directory*	This points to a directory where image files are physically stored on disk. If any column definition has an option indicating it contains image file names, then the files in the column are searched for first as named, and then in this directory. If the image can be found, then the image tag can be generated with width and height parameters, which normally speeds up the page rendering speed by the browser.
imageurl=*url-prefix*	When image tags are generated in a column, the *url-prefix* is placed in front of the file name. This allows the Web server to map the name to a physical location on the server.

complete=*text*
incomplete=*text*
	One of these values is placed in the "$status" global string at the end of each page, depending on whether the job is complete or not. You can then place the value in the HTML template files by embedding the tag "[$status]" in the template.

HTML Output Templates

As companies develop Internet and Intranet strategies, they should employ standard formatting conventions to their HTML documents. HTML-formatted reports should likewise follow these conventions, so UnForm supports the use of HTML template files.

UnForm looks for these files in the UnForm directory, each named in the parameter file "ufparam.txc" or "ufparam.txt". UnForm is distributed with a standard parameter file and standard HTML template files. To customize these for your site, copy "ufparam.txt" to "ufparam.txc", then copy the template files to new names and reference those names in the new "ufparam.txc" file.

The names to use are specified in the "[html]" section of the parameter file, and are coded as "toc=*tocfilename*", "page=*pagefilename*", "both=*bothfilename*", and "frame=*framefilename*". In each of these files, place the text "[$toc]" where the table of contents should be placed, and "[$page]" where the page table(s) need to be placed. In the case of a frame template, the two markers are used for placement of URL links to the table of contents document and the page document(s), respectively.

UnForm determines which template files are used based on the style being used for the output. If there are separate table of contents and page documents, then the *tocfilename* and *pagefilename* are both used. If the table of contents and the pages are in the same document, then the *bothfilename* is used. This file should contain both [$toc] and [$page] tags. If frame output is used, then the *framefilename* is used for the primary document, and the *tocfilename* and *pagefilename* files are used for the target documents.

In addition to the required [$toc] and [$page] tags, you can also reference other pre-defined tags: [$title], [$date], [$time], and [$status], as well as any global strings that you define in prepage{} or prejob{} code blocks. These global strings, generated by the STBL() or GBL() functions, are embedded in the document by placing the name in square brackets anywhere in the template.

One special note: If you wish to customize the date and time masks used by UnForm, set DATEMASK$ and/or TIMEMASK$ in the prejob{} code block to the desired format based on the BBx DATE() function.

The default HTML template for a page (page=*filename*) looks like this:

```
<html>
<head>
<title>[$title]</title>
</head>
<body bgcolor=#e0e0e0>
<h3><center>[$title]</center></h3>
<hr>
[$page]
<hr>
<center><small>
&copy;1997 by Synergetic Data Systems Inc.<br>
```

```
All rights reserved.
</small></center>
</body>
</html>
```

The default template for an independent table of contents (toc=*filename*) looks like this:

```
<html>
<head>
<title>[$title]</title>
</head>
<body bgcolor=#e0e0e0>
<center>
<h3>Table of Contents</h3>
<strong>[$title]</strong>
</center>
<hr>
[$toc]
<p>[$status]
<hr>
<center><small>
&copy;1997 by Synergetic Data Systems Inc.<br>
All rights reserved.
</small></center>
</body>
</html>
```

The default template for a combined style (both=*filename*) looks like this:

```
<html>
<head>
<title>[$title]</title>
</head>
<body bgcolor=#e0e0e0>
<h3><center>[$title]</center></h3>
<center>[$toc]</center>
<hr>
[$page]
<hr>
<center><small>
Run on [$date] [$time]<p>
&copy;1997 by Synergetic Data Systems Inc.<br>
All rights reserved.
</small></center>
</body>
</html>
```

The default template for a frame style (frame=*filename*) looks like this:

```
<html>
<head><title>[$title]</title></head>
<frameset cols="25%,*">
 <frame name="toc" src="[$toc]">
 <frame name="page" src="[$page]">
</frameset>
</html>
```

HTML Rule Sets

Like PCL rule sets, HTML rule sets are stored in a text file. Each set is headed by a unique name in square brackets:

[AgingReport]
keywords…

UnForm selects a rule set to use based on either the "-r *ruleset*" command line option, or **detect** keywords in each rule set. **Detect** keywords cause UnForm to scan the first page of input, then search for a match where all **detect** keyword(s) for a given rule set match the contents of the page.

Once a rule set is selected, UnForm begins processing each page of text using the rules specified. Each page is first stripped of any PCL escape sequences so that just text remains, then the array of text rows is converted to HTML based on the rules. This HTML is then placed in the output according to the style of output defined by the rule set.

If no rule set is selected, then UnForm will process each page as plain text, using HTML <pre> and </pre> tags, with horizontal rules between pages (where form-feeds occur in the input).

The following keywords are identical in use and function with printer rule sets:
- cols
- const
- detect
- page
- rows

The **hline** and **vline** keywords are identical, except that they *always* perform an erase of the horizontal and vertical lines found.

Keywords unique to HTML generation are defined on the following pages.

BORDER

Syntax

border=*value*

Description

The tables generated by UnForm for each page will normally have borders, and will therefore set the table border option to 1: <table border=1 ...>. If you would prefer a different border setting, define it with this keyword.

See also the **otheropt** and **width** keywords.

COLDEF

Syntax

1. [coldef | ccoldef] *col*, *cols*, *options*
   { *code block* }

2. coldef "*text* | ~*regexpr", coloffset, cols, options*
   { *code block* }

3. coldef "*text* | ~*regexpr", coloffset, "to-text* | ~*to-regexpr", to-coloffset options*
   { *code block* }

Syntax 1 defines an absolute column region. **coldef 30,21** for example, would define a column region from column 30 for 21 columns (30-50). If the "ccoldef" syntax is used, then *col* is the starting column, and *cols* is the ending column. **ccoldef 30,50** would define the same region as above.

Syntax 2 defines a region based on a search for a starting point. For each *text* value or *regexpr* (regular expression) found, the region will begin at the column *coloffset* from the point found, and extend for *cols* columns. For example, **coldef "Customer total",-1,52** will create the region from 1 column before the occurrence of "Customer total", and extend the region for 52 columns.

Syntax 3 defines the region based on two searches, one to find the starting column, one to find the ending column to the right of the starting point. In both cases, the column position is adjusted for the offset. **coldef "Current",-1,"30-Days",-1** would define a region starting one column before the word "Current", extending to one column before the word "30-Days". If just the first string is found, then all columns from there to the last are specified. If just the last string is found, then all columns from the first through there are specified. For this reason, be sure that any absolute column regions are specified first.

Description

Column definitions are used to define columns within a row definition. Each column definition becomes a table cell (<td>…</td>), with each row in the column being separated by a line break (<br>). There can be up to 255 column definitions within any given row definition. Any given column will be formatted based on the first **coldef** keyword that applies to it. Columns not so defined will be displayed as mono-spaced text, using the HTML <pre> and </pre> tags.

Each column definition can define attributes that will apply to the text and cell formatting, and optionally can have a code block associated with it to add custom Business Basic coding to the data in the column.

Options are comma-separated lists of words and parameters. The options available in the column definition include:

Option	**How it gets applied**
bgcolor=*#rgb*, bgcolor=*color*	Cell gets a bgcolor=*value* attribute to control the background color. The color can be expressed as an #rrggbb hexadecimal value or as a color name supported by the target browser, such as red, blue, white, etc..
blink	Text gets <blink> attribute.
bold	Text gets <b> attribute.
bottom, top, middle	Cell gets "valign=*value*" attribute to control vertical justification. The default is "top".
center, left, right	Cell gets "align=*value*" attribute to control horizontal justification. The default is "left".
color=#rgb, color=*color*	Text gets <font color=*value*> attribute. The color can be expressed as a #rrggbb hexadecimal value or as a color name supported by the target browser, such as red, blue, white, etc..
font=*font*	Text gets <font face=*font*> attribute. Several modern browsers support this, though the *font* typeface selected may not be available on all clients.
hdr=*html text*	The top of the column gets the *html text*, followed by a line break tag. Use this option to replace top of page column headers with "in cell" column headers.
hdron=*hdron text* hdroff=*hdroff text* hdrtd=*hdrtd text*	The column header, if defined with hdr, gets these values in its <td *hdrtd*>*hdron* hdr value *hdroff*</td> structure. Be sure to turn off any *hdron text* HTML tags in *hdroff text*.
italic	Text gets <i> attribute.
image	Text is assumed to be file names that are image files, and gets treated as an <img> tag. The ufparam.txc|t file values for imagelib and imageurl are used for image processing. The imagelib value is used to locate files on the web server's file system in order to calculate width and height values (.gif and .jpg files only.) The imageurl value is prefixed to the report data when constructing the <img src="*image URL*">.
ltrim, rtrim, trim	These three mutually exclusive options will cause UnForm to left, right, or left and right trim the text of the column when generating the HTML cell text. By default, any spaces in the data for the cell remain in the output. Use of this option may save some disk storage space and document transmission time.
noencode	If this option is present, then the text is not encoded for HTML markup entities. This should only be used if you know that the text contains valid HTML coding.

Option	**How it gets applied**
otheropt=*options*	The table cell gets additional attributes not otherwise specified by the other options.
size=*n*	Text gets <font size=*n*> attribute. Size ranges from 1 to 7, with 3 being considered a "normal" size.
suppress	If this word is present, then column data gets set to null.
underline	Text gets <u> attribute.

Code blocks are optional definitions associated with any given column definition. With a code block, it is possible to manipulate the text of each row in the column. A typical use of this capability might be to convert the plain text to hyperlinks, so that a column of part numbers could be linked to pages in a catalog, for example. Code blocks begin just after the opening brace "{", can extend as many lines as required, and end with a closing brace "}".

The code block is executed for each row of the column. As the code starts, the following variables can be used:

Variable	**Description**
attr.align$ attr.bgcolor$ attr.blink attr.bold attr.color$ attr.font$ attr.italic attr.otheropt$ attr.size$ attr.underline attr.valign$	The attr$ variable is a string template that defines the attributes to apply to the text or cell. These values match those defined above in the Options. Numeric values can be set to 0 (false) or 1 (true). String values can be set to any valid value for that attribute.
colofs	The column offset from the left edge of the text. If the column region is from column 21 through 40, then colofs will be 21. This should be treated as a read-only value.
cols	The number of columns in the region. Read only.
row	The row number within the current region, from 1 through the last row in the region. With each execution of the subroutine, the row will increase by 1. Read only.
row$	The text of the current row within the region. This can be manipulated by the code.
rowofs	The position of the current row, relative to the whole page. If you need to refer to data in some other column of the current row, use rowofs. Read-only.

Functions available for your use, in addition to any intrinsic Business Basic functions, include:

Function	**Description**
get(*col*,*row*,*cols*)	Returns text from the page, given the column, row, and cols parameters.
htmencode(*text$*)	Returns *text$* after converting HTML entities into displayable versions.
set(*col*,*row*,*cols*,*text$*)	Sets *text$* into the page at the given column, row, and columns.
urlencode(*text$*)	Returns *text$* after URL encoding to make it suitable for inclusion in a hyperlink.

COLWIDTH

Syntax

colwidth=*text*

Description

When UnForm generates a table for each page of a document, it defines a standard column cell width so that text that lines up vertically in the report will remain lined up in the HTML version. UnForm generates an initial single row of individual cells, using *text* as the cell width, as used in the HTML tag "<td width=*text*>".

If a *text* value, such as a pixel count or other valid HTML cell width is specified, then UnForm will use that value when defining the initial column cell sizes for each page.

FRAME

Syntax

frame=y | yes | n | no

Description

The **frame** keyword can be used in conjunction with the **multipage** keyword to control the presentation of the report. Without these options, UnForm will produce a single file (named with the **output** keyword or –o command line option, or to stdout), containing an HTML table for each page of output from the source file. With the **multipage** keyword, UnForm will produce unique files for each page of output, plus a table of contents page (whose format is controlled by the **toc** keyword). If frame is set to "y" or "yes", then an additional frame file is created for the browser to view the table of contents constantly while viewing the report pages.

The output filename generated is for the frame file if frame is set to "y" or "yes", and the table of contents file if frame is not present or is set to any other value.

This keyword is ignored if there is no *filename* specified for the output.

HDRON, HDROFF, HDRTD

Syntax

hdron=*value*
hdroff=*value*
hdrtd=*value*

Description

When a coldef **hdr=*text*** option is present, UnForm will add *text* to the top of the column, in a separate cell. In order to make a column-heading stand out, it may be desirable to give it attributes that are distinct from the column text. These keywords define HTML text attributes to add before and after any column header. **hdrtd** applies <td *value*> to the cell tag, while **hdron** and **hdroff** apply to the heading text. Values for individual row groups can be specified in the **rowdef** or **coldef** keywords.

For example, **hdron=<small><b>** and **hdroff=</small></b>** would make column headings small and bold.

Be sure to close any tags in the **hdron** value with corresponding tags in the **hdroff** variable.

LOAD

Syntax

load *filename*

Description

The **load** keyword is used to load a secondary text file into the rule file at parsing time, at the position of the **load** keyword. This provides the ability to maintain separate text files for the definitions, grouped in any manner desired. For example, a common set of options for all reports could be defined in a second file, and each report could reference that file.

UnForm will try to open the file first as named, then in the UnForm directory if it is not found. Note that the prefix setting, if present, in UnForm's config.unf file can be used to affect file searching.

Example:

```
[Report1]
load "stdoptions.txt"
```

MULTIPAGE

Syntax

multipage=y | yes

Description

If multipage is set to "y" or "yes", UnForm will generate a different document file for each page of output. The pages will be named *filename.pagenum*.htm, with *pagenum* being the sequential page number of the report.

A table of contents will automatically be generated as well, with each link in the table of contents referencing the proper document name. The table of contents file will be named one of two names: *filename*.toc.htm if a frame structure is being generated, or *filename*.htm if not. When no frame is generated, then the table of contents document becomes the base document for the output.

This keyword is ignored if there is no *filename* specified for the output.

NULLROW

Syntax

nullrow=y | yes

Description

If this value is set to "y" or "yes", UnForm will print undefined row sets as mono-spaced text, using HTML <pre> and </pre> tags. By default, UnForm will suppress any rows that have not been allocated with **rowdef** keywords.

OUTPUT

Syntax

output "*filename*"

Description

If no "-o *filename*" is specified on the command line, UnForm will use the file *filename* specified here. Use this keyword to specify a default output location for any given report.

UnForm automatically adds a ".htm" extension to *filename*.

OTHEROPT

Syntax

otheropt "*table-options*"

Description

When UnForm generates a table for each page of the document, it establishes border and width options for the table tag: <table border=*border* width=*width*>. If additional options are desired, specify them with this keyword. If present, the table tag is generated like this:

<table border=*border* width=*width table-options*>

See also the **border** and **width** keywords.

PAGESEP

Syntax

pagesep "*html code*"

Description

If a single document is generated for all pages of output (multipage is not set to "y" or "yes"), then UnForm will place a paragraph tag (<p>) between each page. If something other than a paragraph tag is desired, then specify the HTML code in the **pagesep** keyword.

The **pagesep** value can contain global string values generated from code blocks by referencing the string value name inside square brackets.

For example: **pagesep "<p><hr>[pagehdr]"** would generate a paragraph tag plus a horizontal rule, followed by the value in the global string "pagehdr", defined with the STBL() function in a prepage{} or prejob{} code block.

PREJOB, PREPAGE, POSTJOB, POSTPAGE

Syntax

prejob | postjob | prepage | postpage {
code block
}

Note: the opening brace "{" needs to be on the same line as the keyword. The closing brace may follow the last statement, or be on the line below the last statement.

Description

These keywords are used to add Business Basic processing code to the document generation process. They represent four different subroutines that UnForm executes at specific points during processing. The *code block* can be an arbitrary number of Business Basic statements; the total number of statements in all code blocks can be about 6,000 (or less, depending on program size limits imposed by the run-time environment).

- **prejob** executes after the rule set has been read, and after the first page is read, but before any printing takes place. Use this code to open files or databases, prepare SQL statements or string templates, create user-defined functions, and initialize job variables.
- **postjob** executes after the last page has been printed. Use this to close out your logic, such as adding totals to log reports. There is no need to close files, since UnForm will RELEASE Business Basic.
- **prepage** executes after each page is read, but before any printing takes place. Use this to gather data associated with any page, or to modify the content of the text if you need such modifications to apply to all copies.
- **postpage** executes after the last copy of each page has printed.

Any valid Business Basic programming code can be entered, including I/O logic, loops, variable assignments, and more. Program to your heart's content. UnForm will add extensive error handling code within your code, and report syntax errors to the error log file or a trailer page.
You may use the following variables and functions in your *code block*:

- **text$[all]** is a one-dimensional array of the text for the page. For example, text$[2] is the second line of the page.
- **mid(*arg1$,arg2,arg3*)** (or fnmid$(*arg1$,arg2,arg3*)) is a function that safely returns a substring without generating an error 47 if the value in *arg1$* isn't long enough to accommodate position *arg2* and length *arg3*.
- **get(*col,row,length)*** (or fnget$(*col,row,length*)) is a function that safely returns text from the text$[all] array, without substring or array out-of-bounds errors.

- **set(*col,row,length,value$*)** (or fnset$(*col,row,length,value$*)) is a function that places *value$* in the text$[all] array at the place indicated. It returns *value$*.
- **err=next** may be used for any err=*label* option in any function or statement, in order to force UnForm's error trapping to ignore an error. You may, of course, name your own err=*label* if desired.

When using variables and line labels, you should avoid using any values that begin with "UF_". UnForm reserves all such variables and labels for its own use. You may use a backslash (\) at the end of a line to continue the statement on the next line. Lines prefixed with "#" are not added to the code.

A discussion of programming in Business Basic is outside of the scope of this manual. If your needs require programming, then it would be advisable to hire a professional Business Basic programmer, acquire training for a technical member of your staff, or contract with SDSI for your needs.

Column definitions can also have code blocks, which are executed as each row of a column definition is generated. See the **coldef** keyword for more information.

ROWDEF

Syntax

1. [rowdef | crowdef] row, rows, *options*
   { *code block* }

2. rowdef "*text* | *~regexpr", rowoffset, rows, options*
   { *code block* }

3. rowdef "*text* | *~regexpr", rowoffset, "to-text* | *~to-regexpr", to-rowoffset options*
   { *code block* }

Syntax 1 defines an absolute row region. **rowdef 5,3** for example, would define a row region starting with row 5, and extending 3 rows down (5-7). If the "crowdef" format is used, then *row* is the starting row, and *rows* is the ending row. **crowdef 5,7** would define the same region as **rowdef 5,3**.

Syntax 2 defines a region based on a search for a starting row that contains the text or matches the regular expression. For each *text* value or *regexpr* found, the region will begin at the row *rowoffset* from the point found, and extend for *rows* rows. For example, **rowdef "Customer total",0,1** will create a region from each row containing "Customer total" (0 offset is that row), and extending for 1 row (just that row).

Syntax 3 defines the region based on two searches, one to find the first row, one to find the ending row below the starting row. In both cases, the row used for the region is adjusted for the offset. **rowdef "Customer:",1,"Customer:",-1** would define a region between each occurrence of the text "Customer:". If just the first string is found, then all rows from there to the last are specified. If just the last string is found, then all rows from the first through there are specified. For this reason, be sure that any absolute regions are specified first.

Under format 3, if the last string is not found, UnForm will continue that row definition on the page following the first unallocated row at the time this row definition is evaluated on that page.

Description

Row definitions are used to define sets of rows for which a given group of column definitions would apply. Each row definition defines a group of rows that will be presented within a single table row (<tr> ... </tr>). Under any given row definition, place the column definitions (**coldef** keywords) that will be used to format the rows.

For example, an A/R Aging Report might contain a report heading, column headings, one or more customer headings, and, under each customer heading, one or more detail lines. At the end of the detail lines would be customer totals. This report would have five row definitions, for each type of row: report heading, column heading, customer headings, detail lines, and totals. Each of these types of rows will have its own set of column groups (or in some cases, no column groups at all, allowing simple mono-spaced presentation.)

There can be up to 255 row definitions within any rule set.

Each row definition can define attributes that will become defaults for the text and cell formatting of all the column definitions. Additionally, row definitions can define an option called "suppress", which causes UnForm to suppress the display of the row region. A comma separates each option.

Option	**How it gets applied**
bgcolor=*#rgb*, bgcolor=*color*	Cell gets a bgcolor=*value* attribute to control the background color. The color can be expressed as an #rrggbb hexadecimal value or as a color name supported by the target browser, such as red, blue, white, etc..
blink	Text gets <blink> attribute.
bold	Text gets <b> attribute.
bottom, top, middle	Cell gets "valign=*value*" attribute to control vertical justification. The default is "top".
center, left, right	Cell gets "align=*value*" attribute to control horizontal justification. The default is "left"
color=#rgb, color=*color*	Text gets <font color=*value*> attribute. The color can be expressed as an #rrggbb hexadecimal value or as a color name supported by the target browser, such as red, blue, white, etc..
font=*font*	Text gets <font face=*font*> attribute. This is supported by several modern browsers, though the *font* typeface selected may not be available on all browser clients.
hdr=*html text*	The top of the column gets the *html text*, followed by a line break
 tag. Use this option to replace top of page column headers with "in cell" column headers.
hdron=*hdron text* hdroff=*hdroff text* hdrtd=*hdrtd text*	The column header, if defined, gets placed in a cell with <td> attributes specified *hdrtd text*, and text attributes *hdron text* and *hdroff text*. Be sure to turn off any *hdron text* HTML tags in *hdroff text*.
italic	Text gets <i> attribute.
noencode	If this option is present, then the text is not encoded for HTML markup entities. This should only be used if you know that the text contains valid HTML coding.
otheropt=*options*	The table cell gets additional attributes not otherwise specified by the other options.
size=*n*	Text gets <font size=*n*> attribute. Sizes range from 1 to 7, with 3 being considered a "normal" size.
suppress	The rows are not displayed.
tr	Each row in the row group gets a <tr> tag, ensuring that

Option	**How it gets applied**
	column definitions, even if they contain data values of varying height, will remain horizontally contiguous. If the cells contain only text, this is generally not required, but if some cells contain images, this keyword will likely be required.
underline	Text gets <u> attribute.

TITLE

Syntax

title "*title text*"

Description

The title for any report can be defined in the rule set with this keyword. Once defined, anywhere in HTML output templates that the tag "[$title]" is placed, this text will be substituted.

TOC

Syntax

toc=y | yes | li | list | sh | short

Description

If this keyword is set to "y" or "yes", UnForm will generate a simple table of contents by constructing hyperlinks to each page generated. The hyperlinks are placed either at the top of the document, in a separate main document, or in a document referred to as the table of contents in a frame.

The following templates use a table of contents. Templates refer to files in the UnForm directory, and are referenced in the parameter file under the "[html]" section: "both=" and "toc=". In each case, the placement of the table of contents is based on the placement of the tag "[$toc]" within the template file.

The text displayed for each hyperlink is generated from the "pagenum=" item of the "[html]" section of the parameter file (ufparam.txc or ufparam.txt.) This text can also be generated by Business Basic code in the prepage{} or postpage{} code blocks, by setting the string variable "toc$" to the value desired.

If the keyword is set to "li" or "list", then the hyperlinks are created within an HTML unordered list (<ul> ... </ul>), and will normally be displayed as a bullet list.

If the keyword is set to "s", "sh", or "short", then the table of contents links consist of just the pagenum descriptor followed by each page number, with no line breaks or bullets. In this case, any code that sets the value of toc$ is ignored.

This keyword is ignored if there is no *filename* specified for the output.

WIDTH

Syntax

width=*value*

Description

The tables generated by UnForm for each page will normally occupy the entire width of the page, and will therefore set the table width to 100%: <table width=100% ...>. If you would prefer a different width setting, define it with this keyword. Be sure that if the value is a percentage of the screen, it has a trailing "%".

See also the **otheropt** and **border** keywords.

Sample HTML Rule Set

Below are sample rule sets defined in the sample rule file, samphtml.rul. The sample text input files used by UnForm for the PCL output examples are redefined here for HTML. Comments are interspersed in the rule sets to help clarify which keywords perform which tasks.

Aging Report Sample

To produce this aging report sample to a file, execute the following command:

uf80c -i sample3.txt -o aging.htm -p html –f samphtml.rul

You can substitute a different path/file name for "aging" to produce the HTML file elsewhere, such as in the HTML document tree of your Web server.

The form is called "aging" to distinguish it from other rule sets. If the "-r aging" option is used on the command line, then this set will be used.

```
[aging]
```

A detect statement identifies a report as the one defined by this rule set. If no "-r ruleset*" option is used on the command line, then this detect statement will be evaluated. If the text "Detail Aging" appears in any column on row 2, this rule set is used.*

```
detect 0,2,"Detail Aging"
```

The HTML output will produce 132 columns and 66 rows per page.

```
cols 132
rows 66
```

Any text consisting of 3 or more dashes will be erased. This removes all the dashed underlines at customer totals. There are other ways to accomplish this, including defining a row set and using the suppress option, or using a prepage{} code block to erase such text from the text$[] array.

```
hline "---"
```

The title used in HTML output for this report will be "Aging Report".

```
title "Aging Report"
```

If this line were not commented out (with the #), then anytime this rule set was used and no "-o filename*" was present on the command line, the output would go to "/tmp/aging.htm."*

```
#output "/tmp/aging"
```

This report will be generated in multiple files (one per page), with a table of contents page, and with an HTML frame construct.

```
multipage=y
toc=y
frame=y
```

*Between each page will be an HTML <p> tag (a paragraph separator). Any HTML text could be supplied, including references to global strings inside square brackets ([*variablename*]). The hdron/hdroff keywords supply HTML codes to place before and after any column definition headings, defined with the hdr=*text *option in the coldef and rowdef keywords.*

```
pagesep <p>
hdron=<i><b>
hdroff=</b></i>
```

This rowdef keyword defines a row set from row 1 for 5 rows. All column definitions within this row will default to a background color RGB hex value of FFE0E0 (lots of red, high green and blue content).

```
rowdef 1,5,bgcolor=#ffe0e0
```

For the above row set, there are three column sets: 1 through 10, 11 through 110, and 111 through 132. The columns are left, center, and right justified, respectively. Otherwise, except for the background color, the browser will use its default values for displaying the data.

```
coldef 1,10,left
coldef 11,100,center
coldef 111,22,right
```

This row definition causes UnForm to suppress display of rows 6, 7, and 8 (the column heading information). The rule set will define the column headers as necessary in other row sets.

```
rowdef 6,3,suppress
```

Each customer has a heading line, distinguished by the occurrence of a phone number in those rows. The initial quoted value "~\(...-...-....\)" instructs UnForm to search for a regular expression match that looks like a U.S. phone number in parentheses. From any and all such rows, it will start at 0 rows up or down, and continue for 1 row. This defines those and only those rows that contain the phone numbers. Columns defined for those rows will be bold, with blue text on a white background. As no columns are defined under this row definition, UnForm allocates one column set the full 132 columns wide, and applies the row defaults to the text.

```
# Customer header
rowdef "~\(...-...-....\)",0,1,bold,color #0000ff,bgcolor #ffffff
```

The invoice detail lines represent the most complicated of the row definitions, as there are numerous columns with two different formats. We define constants for the two formats (left and right justification being the only difference.) Then the rows are defined as any rows that contain a date structure of 2 characters, a slash, 2 characters, a slash, and 2 more characters. Note that even though some heading rows have this structure, those rows have already been allocated by prior row definitions and won't confuse things here. UnForm searches for any row with a date. Then starting from that row (row offset of 0), it searches for a row that contains 5 dashes. If such a row is found, then the row set goes through the row before (row offset -1) the dashes. If no such row is found, then the row set goes through the last row on the page.

```
# Invoice lines
const LEFT="bgcolor=#e8e8e8,color=black"
const RIGHT="bgcolor=#e8e8e8,color=black,right"
rowdef "~../../..",0,"-----",-1
```

Each invoice line is made up of 13 columns of information. Each has been defined by the ccoldef keyword by starting and ending column values. Additionally, each is given a header value that will appear at the top of the column, and a constant that references other attributes defined earlier in the rule set.

```
ccoldef 1,10,hdr="Invoice",LEFT
ccoldef 11,20,hdr="Due Date",LEFT
ccoldef 21,31,hdr="PO Number",LEFT
ccoldef 32,39,hdr="Ord Number",LEFT
ccoldef 40,45,hdr="Terms",LEFT
ccoldef 46,52,hdr="Type",LEFT
ccoldef 53,64,hdr="Future",RIGHT
ccoldef 65,75,hdr="Current",RIGHT
ccoldef 76,86,hdr="30 Days",RIGHT
ccoldef 87,97,hdr="60 Days",RIGHT
ccoldef 98,108,hdr="90 Days",RIGHT,color=red
ccoldef 109,119,hdr="120 Days",color=red,RIGHT
ccoldef 120,132,hdr="Balance",right,bold,RIGHT
```

The customer totals occur just below the row of dashes at the end of each customer's invoices. This row definition therefore searches for any rows containing 5 dashes, then starts 1 row down, and continues for just 1 row.

```
# Customer totals
rowdef "-----",1,1
```

The first 52 columns make up one column set. The report provides no text, so we include a code block for this column that sets row$ to "Customer Totals:". Note that if this row set contained more than a single row, we could say 'if row=1 then row$="Customer Totals:'. The remaining column sets just apply right justification to the column values.

```
ccoldef 1,52,right
{row$="Customer Totals:"}
ccoldef 53,64,right
ccoldef 65,75,right
ccoldef 76,86,right
ccoldef 87,97,right
ccoldef 98,108,right
ccoldef 109,119,right
ccoldef 120,132,bold,right
```

8.0 ENHANCEMENTS

UnForm 8.0 offers many significant enhancements to its already powerful predecessors. A list of major enhancements is provided below.

Deliver Command

The deliver command and a related deliver() code block function have been added, designed to simplify the task of delivering documents by email or fax. While jobs run, subjobs are executed to produce a document that is then emailed or faxed, depending on the format of a recipient address. Different fax submission methods are supported, including command lines, email, and the internal Microsoft Fax support via the Windows Support Server. A configuration file, deliver.ini, defines how email and fax submissions are processed.

Related to the document delivery capability is address books, maintained with the new Address Book code block object, or the browser interface to the UnForm server.

More details can be found in the Deliver Configuration chapter, and the deliver command and deliver() code block function references.

Cover Pages

Cover pages are used to generate an initial page for a job using a different rule set, allowing for custom cover pages to be generated as part of an UnForm job. This feature can be used when formatting documents for fax delivery, or for other jobs that require an initial page of a different format than the job itself.

Cover pages are defined with the cover command, the -cover command line option, or a set of code block variables, coverset$, coverfile$, and coverargs$. Cover pages can also be turned off in code blocks by setting nocover=1.

Printing Enhancements

True Type Fonts

True Type font support has been added to the pcl, pdf, and postscript drivers. Most True Type fonts that are licensed for embedding and support Unicode characters can be configured in ufparam.txc file's [ttmap] section. See the Server Configuration chapter for information about configuring TrueType fonts, and the Fonts chapter for caveats and recommendations.

Unicode Character Support

In conjunction with the TrueType font enhancement, UnForm can now produce Unicode text output in two ways: via a text command using an embedded TrueType font, or via the textimage() function, which uses Image Magick to generate an image of a Unicode string.

See the Text command for information about Unicode text. There is a ttprint.rul file in the samples directory which can print all characters supported by a configured font.

The second technique for embedding Unicode text is to use a new function, textimage(). This function uses Image Magick to produce an image of text data, using a specified font. For small amounts of printed Unicode text, using images rather than embedded fonts can produce smaller output streams.

See the textimage() function in the Programming Code Blocks, Internal Functions chapter, for more information about this function.

PDF Enhancements

The PDF transparency model modifies shading in images, and the text, box, and shade command shading options.   This allows color AND shading, and allows items below the shaded item to show through, similar to the blending that goes on in PCL output. Transparency can be set on or off by default with the pdftrans=1 or 0 setting in uf80d.ini. It can also be turned on or off via the -pdftrans or -pdfnotrans command line options, or the transparency rule set command. If it is turned off, shading is as in previous releases.

The copies command now produces non-collated copies, like produced in the pcl and postscript drivers. Former versions treated copies and pcopies as identical in pdf output. Note that rule sets that used the copies command during PDF output will produce different page ordering than previous releases. To restore the collated copy order, use the pcopies command.

The protect command now supports 128-bit encryption, by adding the "128" option.

The image command now recognizes a 'page *n*' option, which leverages Ghostscript to generate a full page image of the *nth* page of the PDF file. The normal behavior of the image command and PDF files is to search for the first image element of the file, rather than generating an image of an entire page.

PDF files that contain incremental updates are now supported. Of particular note for this feature are attachments generated by Microsoft Word, with the 'save as PDF' add-in, which automatically adds an empty incremental update.

The micr command is now supported, using the micr TrueType font licensed for inclusion in UnForm. If micr commands should be ignored for PDF output, enclose them in 'if driver laser' blocks.

Support has been added for linearized PDF files as well as those with incremental updates. PDF files up to version 1.4 are now supported, and any later revision files that do not use an internal file structure called an *Object Stream* are also compatible.

Object Oriented Programming Features

New object oriented programming features have been added, offering modern techniques for rule set coding that can streamline code block programming. Several internal objects have been added that simplify many formerly tedious tasks. A chapter is devoted to Object Oriented Coding and to each of the new internal objects. These objects include:

- addrbook - address book management
- binfile - binary file access
- collection - collections of values by index and name
- date - date management
- doclist - library document lists
- http - http/https client for interacting with web servers
- inifile - ini file access
- keyfile - keyed file access
- libraries - library lists
- library - library management
- rac - remote access codes for documents
- search - library search execution
- system - operating system and file system access
- textfile - text file access
- xmlreader - xml document parsing

Code Block Programming Enhancements

In addition to the new object oriented coding techniques, many code block variables and functions have been added in this release. New functions, documented in the Programming Code Blocks chapter, include:

- arrset() updates an array with text values at specified positions, similar to the set() and mset() functions that update the page print stream text array
- basename() returns the file name from a full path
- cdate() converts text data into a date value (a Julian number)
- clientenv() now works in subjobs
- cstrans() translates text between two character sets or symbol sets
- deliver() executes a fax or email delivery of a document
- dirname() returns the directory portion of a full path
- entityencode(), entitydecode() encode and decode HTML entity values in text
- fileext() returns the extension of a filename
- fromuc() translates text from unicode
- getaddress() retrieves an address from an address book
- getcolumn() slices a column of fields out of a series of text lines
- getpaircount() returns the number of name=value pairs in a string

- getpairvalue() parses strings of name=value pairs
- getpatternvalue() locates text patterns in text strings or arrays
- imgx(), imgy() returns an image's width and height
- logwarn() logs a message to the warning messages written to a job's temp/*.err file and to the design tool
- pdfpages() returns the number of pages in a PDF file
- putaddress() writes an entry into an address book
- setlogin() sets login/password values when using the library object
- sqlconnect() connects to SQL databases
- sqlexecute() executes a SQL command on a connection
- sqlfetch() returns rows from a SQL query
- tempfile() generates a temporary file that is automatically removed when the job completes
- textimage() generates an image with unicode text, using Image Magick
- touc() translates text to unicode

New variables include:

- coverargs$ specifies command line options for a cover page subjob
- coverfile$ names a rule file containing coverset$
- coverset$ sets a rule set to use for a cover page
- nocover=1 turns off cover page generation
- noemail=1 turns of the email command
- uf.parent contains the job ID of the parent job, helpful for logging
- uf.login$ contains the -arclogin user ID

Archiving Enhancements

The archive browser interface has been extensively re-written to provide more streamlined access to documents, and to document management. New features include image viewer emailing, direct document access when the library, doctype, and doc ID are known, address book management, custom forms integrated with CGI-driven rule set processing, and powerful saved searches with run-time prompting. A performance improvement in the UnForm images command is noticeable when consolidating marked PDF documents as well.

In addition, new virtual libraries are generated for external users. When an external user (a user with an entity ID) is browsing a library for the first time, a copy is made of the library metadata structures for just records identified with the entity ID of the external user (other users with the same entity ID will share this copy). This copy is maintained when the main library records are modified or deleted. The external users browse this smaller library, providing the full browsing experience, but with only their documents.

Virtual libraries are deleted if unused for 30 days. They are stored in the "ent" subdirectory structure under the library path, and will add to storage requirements, depending on how many entity ID-associated documents are copied, and how many external users login and use a library.

The first-time copy can take a while. The browsing interface warns users of this possibility.

Zebra Enhancements

The lockcols command and the -lockcols command line option prevent recalculation of the cols value.

Support for many new barcode symbologies has been added. See the Zebra barcode command for more information.

Windows Support Server Enhancements

The Windows Support Server now supports compression when communicating with an UnForm server that supports zlib compression (indicated with the uf80c –v command). This can improve performance when using image conversion and scaling or Ghostscript-based PDF to image conversion.

The Support Server is now used when producing barcodes with features not supported in the server-based barcode library. Primarily, this includes 2D barcode support, rotation, and human-readable text support.

The Support Server includes technology to support text parsing from PostScript input streams.

Miscellaneous Enhancements

Multiple archive commands are now supported. Note there is a potential behavior difference if rule sets contain multiple archive commands. Now all execute. In 7.x, only the last one executed.

A new -arclistfmt option, xmlf, adds a base64-encoded embedded file element to the listdocs xml format.

Setting noemail=1 in a code block turns off the email command, allowing code block control over whether or not to execute the email command at the end of the job.

EPS output: -p eps supported to produce an eps file based on the first page, first copy of the job. This allows output from an unform job to be used as an image for a subsequent PostScript job. EPS is also widely supported in document production and imaging programs.

Control of textjob$[all] generation: -notextjob|-textjob command line control, and textjob=1|0 in uf80d.ini [defaults] section to define the default behavior.

Configuration files now support continuation lines using a "\" suffix, which indicates the next line extends the current line. This is particularly useful in the deliver.ini file, and can be used in uf80d.ini well.

The image command now supports a 'page *n*' option when used in the PDF driver with a PDF file named as the image. Ghostscript is invoked to convert the specified page into an image.

In cases where Ghostscript 8.10 or higher is configured, either via the Windows Support Server or in the UnForm server's [drivers] section of uf80d.ini (see pdffitpage=1 notes), the images command is now performance-optimized for both PDF and PostScript output when PDF files are used as images. PDF source images are converted to the required format exclusively with Ghostscript, bypassing the extra processing through Image Magick that was required in previous releases. Black and white PCL output was optimized in previous releases. Note that Postscript 3 (-p ps3) output supports compression of black and white images on systems with zlib support, resulting in lower bandwidth requirements for Postscript printing.

The local timezone in Internet UTC-offset format (i.e. -0800 for Pacific Daylight Time) is automatically resolved based on local system settings, and passed to MailCall when running the email command or email() code block function.

The previous server log file is renamed to uf80d.log.bak upon server restart. The server log format is now a tab-delimited file, to ease log analysis. The Windows UnForm server offers a log viewer with column sorting and filtering.

Windows print driver output is supported using the –o "*winprt*;*name*" option, in conjunction with an installation of Ghostscript, to support any Windows printer. See the –o option in the Command Line Options chapter. Note that this feature is only supported on Windows UnForm servers.

Paper sizes can now be specified in a *width*x*height* format, such as 8.5x11. The format supports specification in centimeters and millimeters (using a cm or mm suffix) as well as the default of inches.

Server-side SQL database access is available via three new code block functions: sqlconnect(), sqlexecute(), and sqlfetch().

A new secure password store has been added, maintained via the browser interface by an administrator. Passwords are stored under an ID, and two functions, setlogin() and sqlconnect(), support a password format of "store:*ID*" to lookup the stored password value at runtime.

Image Manager

Numerous enhancements have been added to the Image Manager, including:

- Improved OCR operations utilizing Microsoft Office Document Imaging
- Custom form definitions, in addition to code driven custom forms

- Server-based script jobs, set value jobs, lookups, and forms, in addition to local jobs
- Support for multiple users in Terminal Server environments
- Group rotation of images at scan/import time as well as via the menu and toolbar
- Support for PDF importing, with automatic Ghostscript-based conversion to tiff
- Numerous script commands and functions for library manipulation and image management
- Uploads are logged in the server log file as line type "scan", with the message containing the library, type, doc ID, sub ID, user, and IP address of the Image Manager
- Scripts can lock the properties grid to prevent accidental editing of values

The Image Manager help file describes the features in detail.

Application Formatted Output

Beginning with version 8.0.03, UnForm accepts PostScript input as a pre-formatted print stream. Using GhostScript and the Windows Support Server (bundled with Windows versions of UnForm and available free for stand-alone installation), the job is parsed into pages that can then be added to by rule sets for cosmetic enhancements, plus access is provided to the text content for document management functions such as archiving and electronic delivery.

More details can be found in the UnForm AFO chapter.

New Learning Resources

A series of new sample rule files have been added that demonstrate specific features of UnForm rule sets, supplementing the already extensive list of samples provided with UnForm.

Caveats

- The **micr** command now supports PDF output. Rule sets that assume PDF output will not have a MICR font line when using the micr command should be modified to suppress it when producing PDF output, either through an 'if driver "laser"' section or by using conditionally assigned variables in the micr command itself.

- PDF output defaults to using transparency when shading text and boxes. The resulting output will show underlying data through the shaded region, unlike previous releases. If this is not desirable, see the **transparency** command.

INDEX