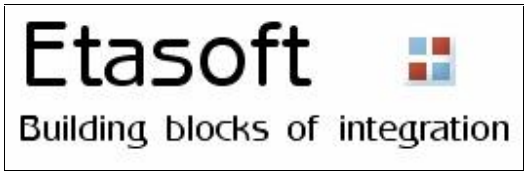


Extreme Processing User's Manual for version 1.x

Main website http://www.etasoft.com
Extreme Processing website http://www.xtranslator.com

Purpose

Extreme Processing is a set of tools for data processing and communication. The backbone of it is XML based job script. Each job contains tasks executed for data send, receive, filtering and other processing. Tasks can trigger other tasks, repeat, execute other jobs, and run based on scheduled time events.

Requirements

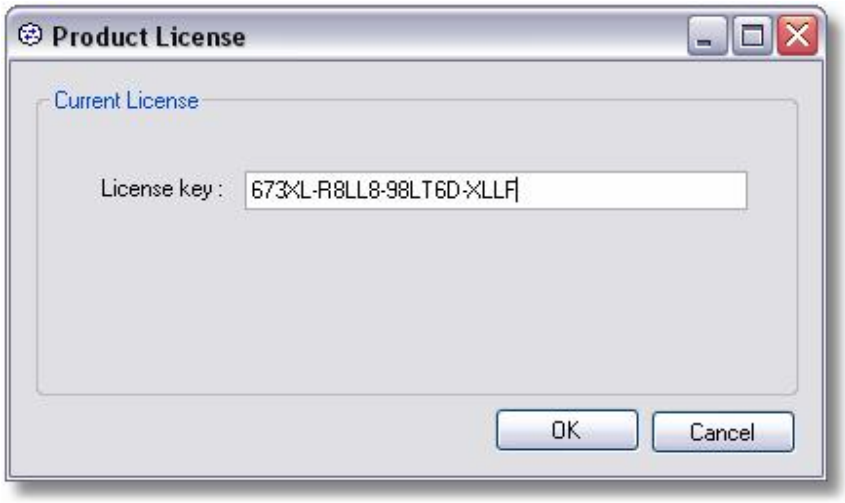
Minimum Requirements	
Software	Windows 2000/XP/Vista/Server 2003
Hardware	Pentium 1GHz, RAM 256Mbt
Recommended	
Software	Windows XP Pro
Hardware	Pentium 1.5GHz, RAM 512Mbt

Terminology and Definitions

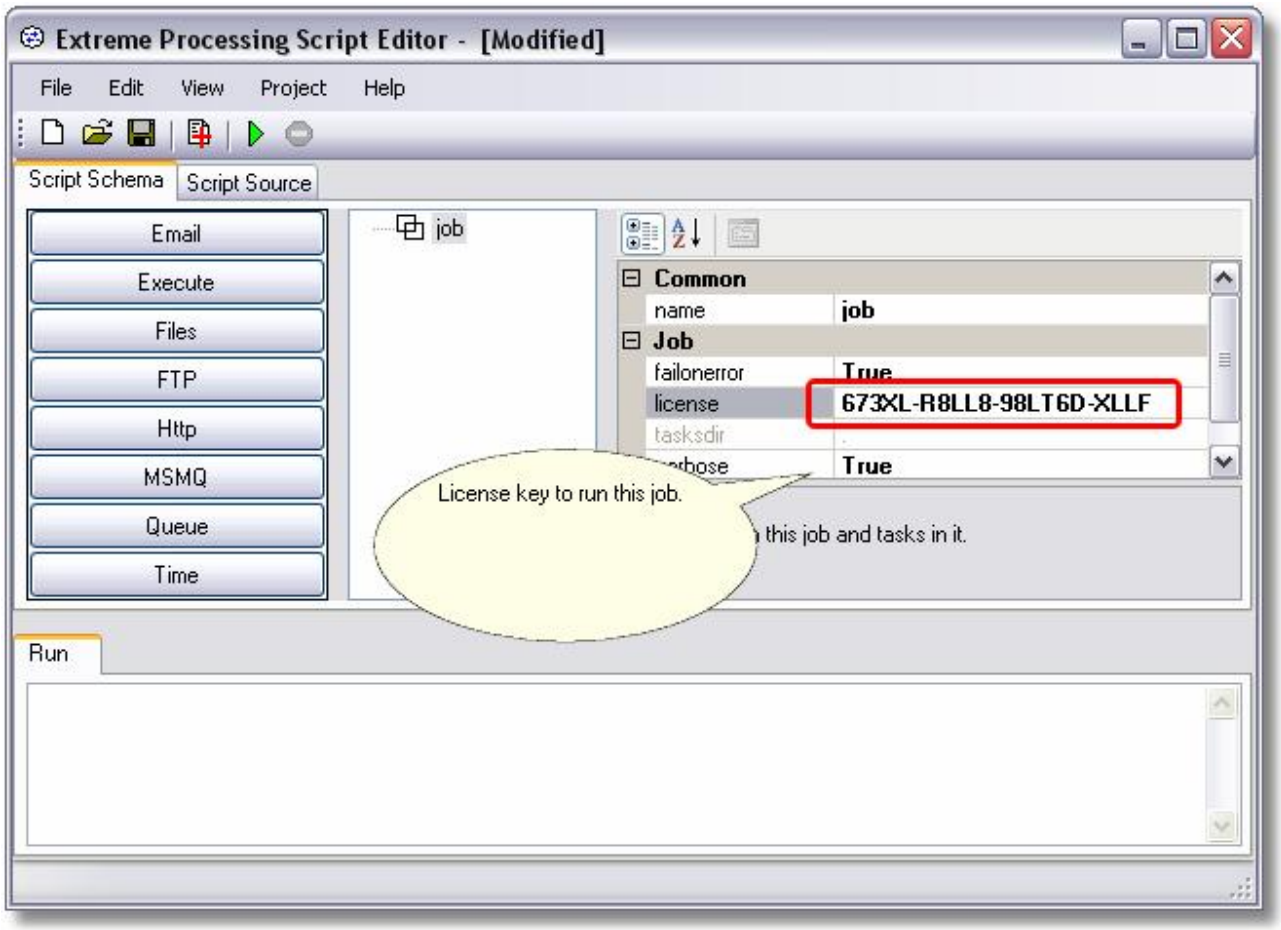
Scripts are XML based job descriptions containing tasks. Each script can contain single root job and number of sub jobs. Sub jobs are executed in separate execution threads.

Licensing

Default package installation comes as time limited trial evaluation license. After fixed number of days starting from the day of the first use, trial evaluation license will expire and most of software features will become disabled. If you will like the product and purchase it, you will receive permanent license key. This new key must be entered in Script Editor and in job property called “license”.



License key should be entered in Script Editor so all future scripts would default to this new license key.



Permanent license key have to be entered after the purchase in old scripts used during trial period if you want to continue using them.

Packaged Tools

Extreme Processing comes with number of tools. Tools are designed to fill three basic needs:
1. Script editing and testing.
You can edit scripts using directly using any XML editor or notepad. However this can be tedious and error prone. We provide graphical Script Editor tool that you can use to add tasks, edit they properties and run them in test environment.

2. Script execution using script execution tools.
You can execute scripts using command line or Windows GUI based tool. Both tools provide simplest way to execute scripts in unattended and automated environment.

3. Script execution via your program using Developer SDK for .NET.

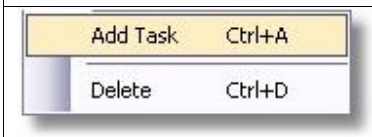
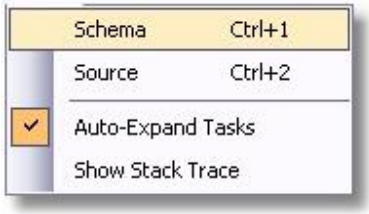
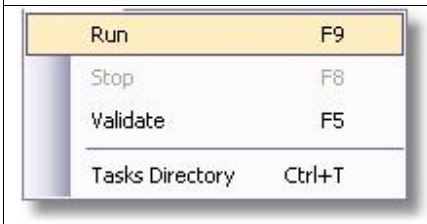
Script Basics

There are some basic rules regarding script creation:

1. Each script has to start with job tag.
2. Each script can contain only one job.
3. Script has to be in valid XML format.
4. Only container tasks can have other tasks nested in them.

Main Menu

Main menu contains standard submenus File, Edit, View, Project and Help. It contains most of functions available. Some functions are duplicated and placed under toolbar buttons, and under right click popup menus.

	"New" menu option is used to start new scripts. "Open" is used to load existing script files. "Save" and "Save As" can be used to save edited script files. "Recent Files" can be used to load recently modified and saved files. "Exit" can be used to close this application.
	"Add Task" can be used to add new tasks from the complete list of available tasks. Available tasks are loaded from "Tasks Directory". They can be changed by pointing "Tasks Directory" to other folder. "Delete" can be used to delete tasks.
	"Schema" can be used to switch editor view to script schema view. "Source" can be used to switch editor view to script source code view. "Auto-Expand Tasks" can be used to set mode when Schema nodes are always expanded. "Show Stack Trace" if turned on will show stack trace if executing job fails.
	"Run" can be used to run current job defined in editor script. "Stop" can be used to stop current job execution. "Validate" can be used to validate script for mandatory values and value ranges. "Tasks Directory" can be used to change directory where tasks are located and loaded from. Default setup loads tasks from current directory.

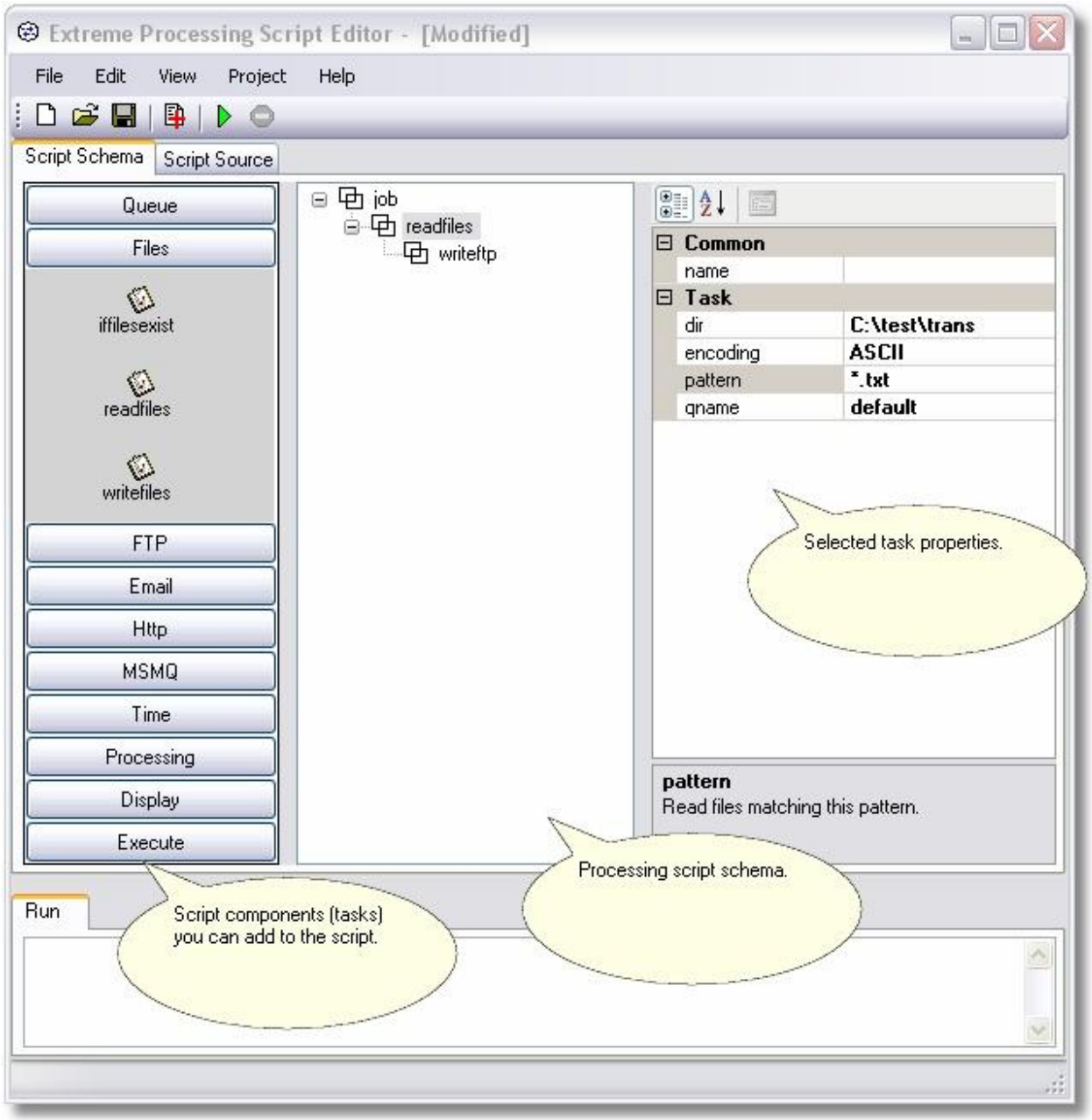
Script Editing

You can edit scripts in any XML editor or notepad. However it is much easier to use graphical Script Editor tool. It combines three most important features:
1. Script schema editing when each task can be added and task properties changed.
2. Script source code editing when actual XML representation of the script could be changed manually in the editor window.
3. Script testing and execution monitoring.
You can use other tools to modify or create job scripts. Scripts have to be valid XML format files, and tasks listed in them should be in tasks directory. Script Editor tool scans directory for available tasks and retrieves they properties.

Tasks are logically divided in to simple tasks and tasks that can contain other tasks (containers). Simple tasks are pure execution entities. They cannot contain other tasks and they do not change script execution flow. Container tasks on other hand can contain other tasks and can change execution flow.
Example of simple non-container task is <pause> task. It simply pauses for defined amount of time. Example of container task is <time>. This task waits for certain time event defined in "schedule" property and executes other tasks nested in it.
Visually simple and container tasks are decorated with different icons:

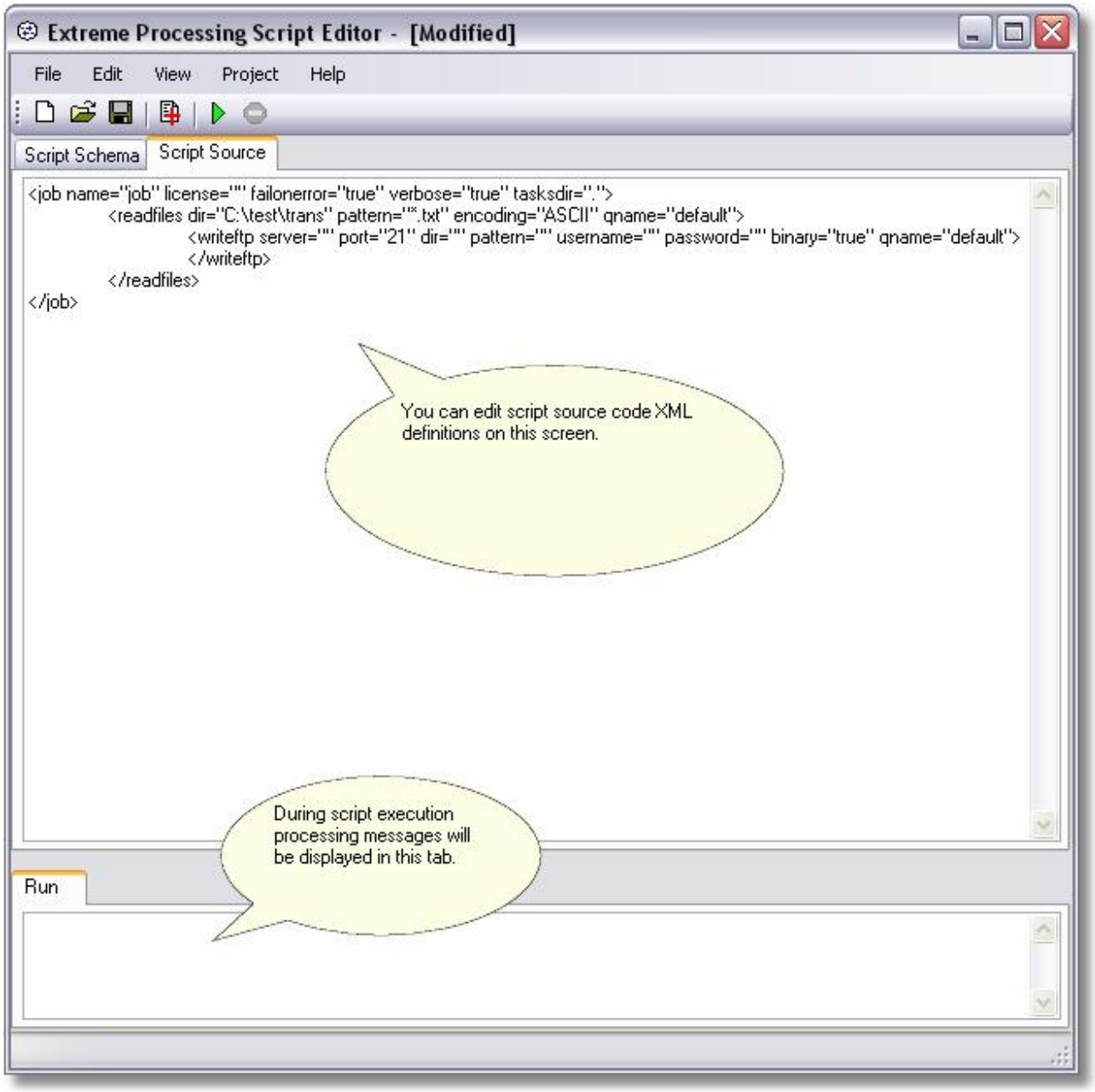
- is used to indicate simple tasks.
- is used to indicate container tasks.

Container tasks can be used in very powerful ways. They can work as IF statements or Loops. Example: <readfiles> task will read files that match certain naming pattern from certain directory, this task will also call other nested tasks, however it will not execute nested tasks if there are no files matching the pattern. This way you can nest <writeftp> task in <readfiles> task and that way files retrieved from the directory will be transferred to FTP server.

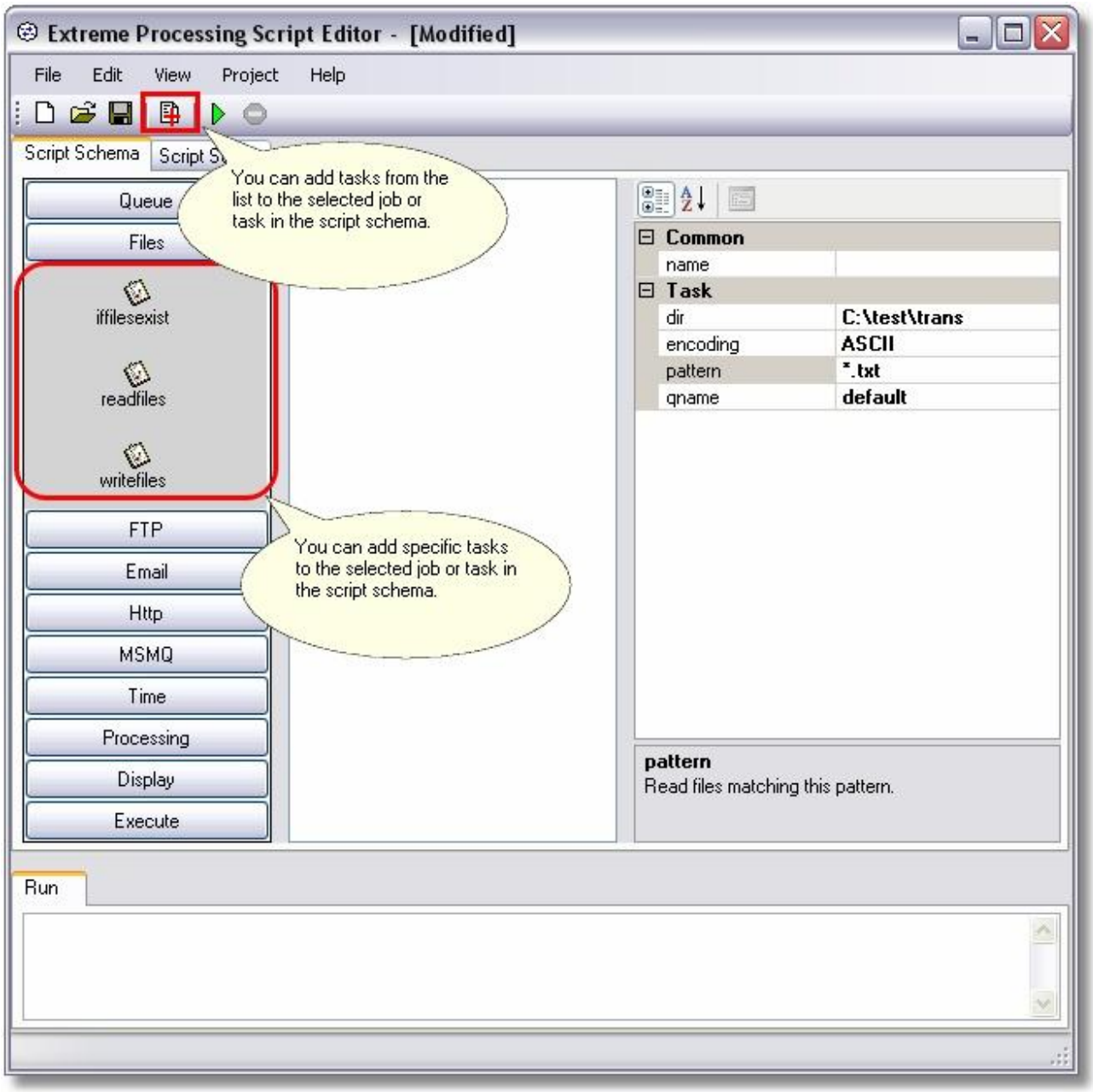


Script Editor main window has sidebar to add new tasks, current job schema and properties for selected task.

All available tasks are refreshed every time Tasks Directory is changed. In order to add or edit or execute tasks Tasks Directory must be set to the directory where DLLs containing tasks are. Default value for Tasks Directory is "." and that means "current directory". In most cases this setup is appropriate unless tasks should be deployed in separate directory than main runtime DLL.



Script source code editor allows direct XML script editing. Once you click “Script Schema” tab back, changes made to source code will move to schema if source code changes are valid. You can right click on this script editing area to copy, paste or select the script.



There are two ways to add tasks to schema: using sidebar or using dialog screen with all tasks listed in the drop down list. Dialog screen has one advantage as you can add tasks at any nested point in the tree, while sidebar only allows addition of the tasks as last nested item.
Most task properties are mandatory. While Script Editor does not enforce them to be set script execution will fail if property is missing value.

This dialog screen allows addition of tasks to any point in the nested tree. You can also edit XML script manually in order to move tasks around but make sure that only container tasks have other tasks nested in them. Also if you copy and paste XML script make sure you copy and paste both XML start and end tags if task has an end tag.

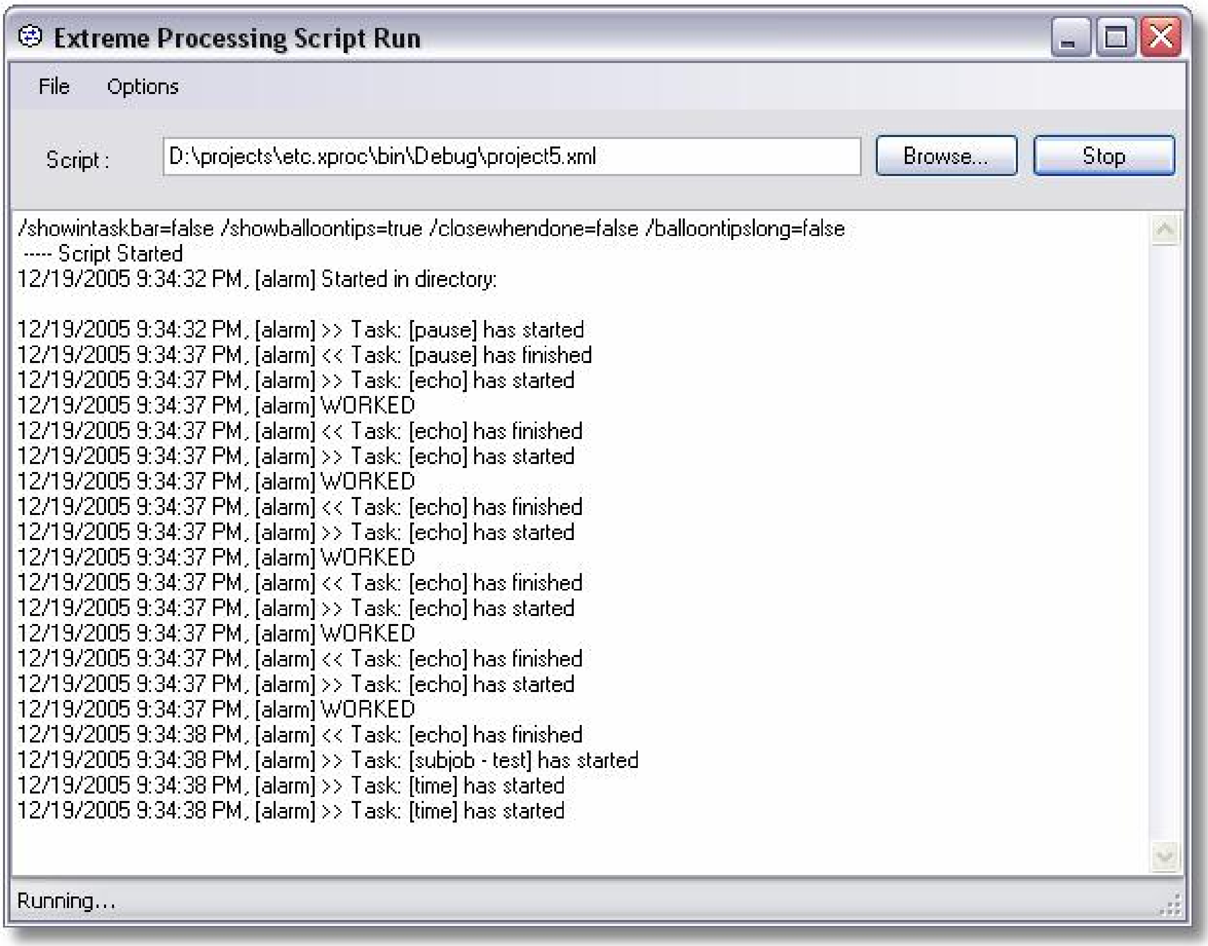
Available Task Packages

All available tasks come in 3 major libraries:

Library	*Description*
Xproc.dll	Basic tasks for general processing.
Iotasks.dll	Input and output tasks for data communication and transfer.
Trtasks.dll	Translator tasks for Extreme Translator map execution.

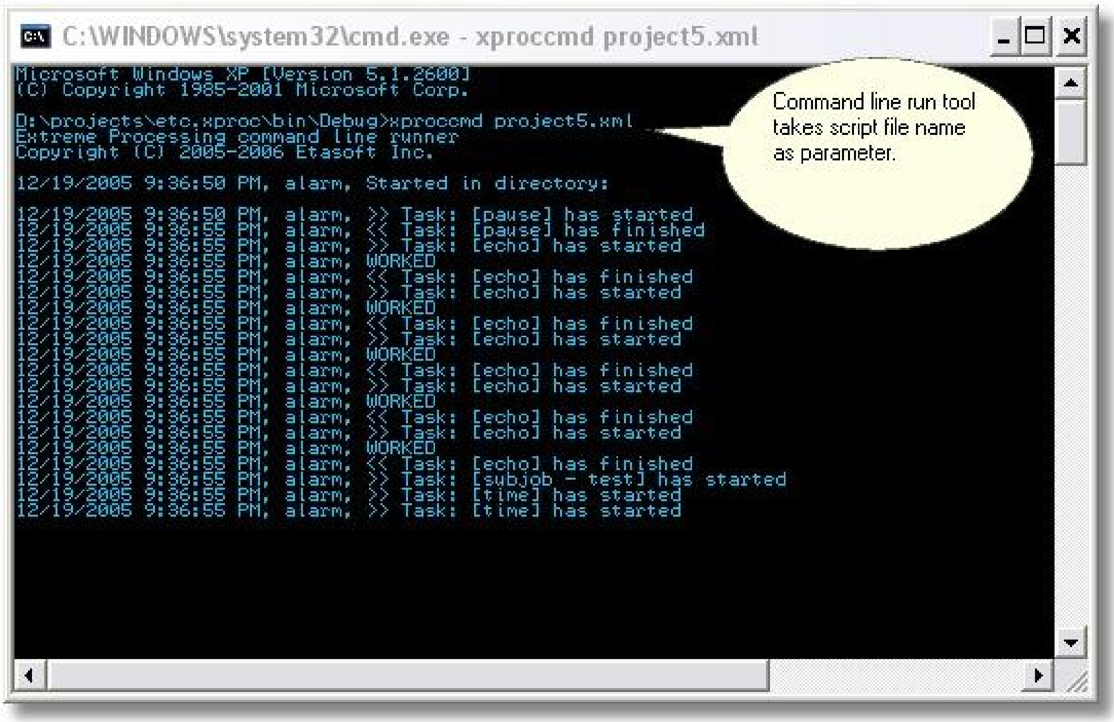
Script Execution

Scripts can be executed unattended using command line and script run tools coming with the package. Scripts can be also executed for testing purposes in the Script Editor. Script Editor execution provides more information as active tasks are highlighted in red. If you have .NET application and would like to run jobs from your application directly take a look at chapter about Developer SDK.



Windows based Script Run tool runs job with job “verbose” property set to true.

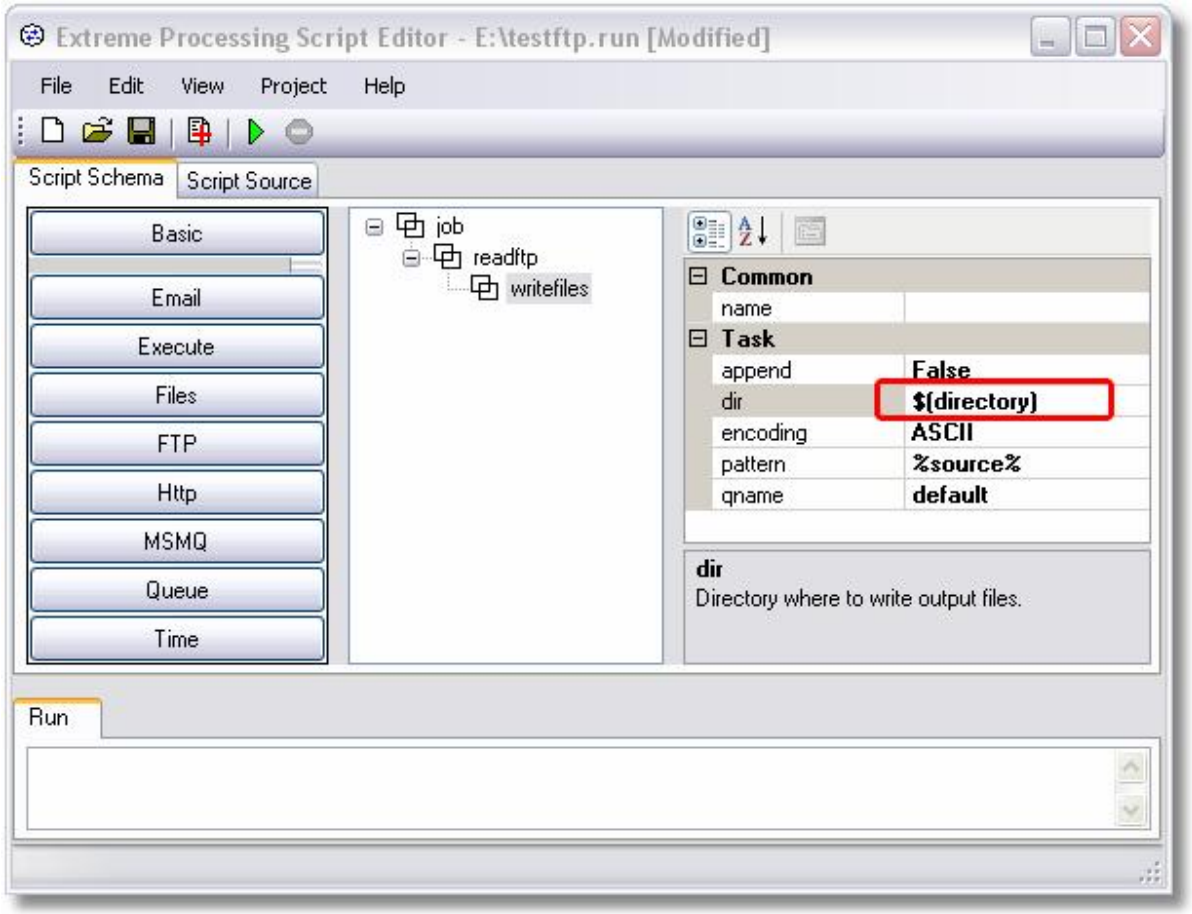
You can pass number of parameters and change Script Run tool options via command line. Accepted parameters:
1. /showintaskbar=[true or false] – if set to true Script Run will always appear in Windows Task Bar.
2. /showballoontips=[true or false] – if set to true will display processing messages as balloon tips near system tray.
3. /closewhendone=[true or false] – if set to true will close itself upon job completion.
4. /balloontipslong=[true or false] – if set to true will show balloon tips until they are explicitly closed.



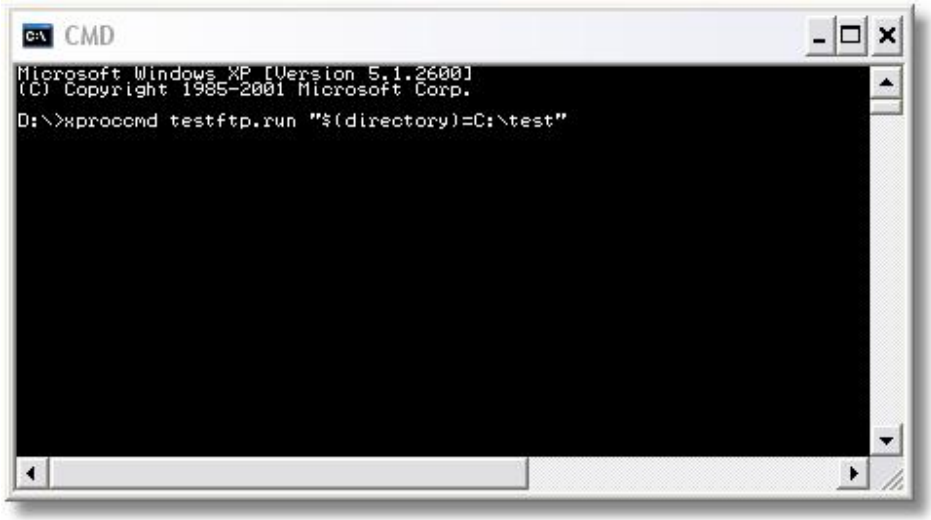
Command line script run tool takes script file name as first parameter. Second parameter is optional and may contain word “-stopiferror” in which case it will stop before closing if error is encountered.

Passing Parameters

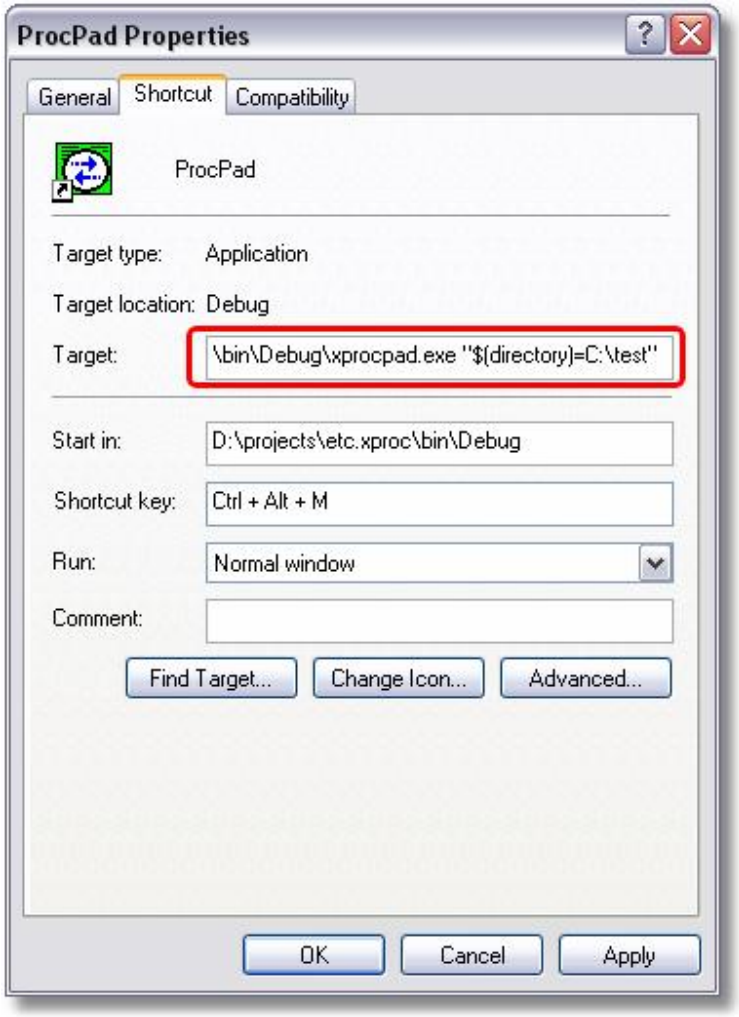
You can pass external parameters to script to supply custom property values for execution. Simple text properties can be passed in via command line or as parameters for job execution. You will need to define script properties as $(_your_variable) in order to be able to substitute them with new values for script to run.



There is example of the property “dir” that accepts parameter called $(directory) passed into the script.



This is example how to pass in external parameter called $(directory) into testftp.run script.

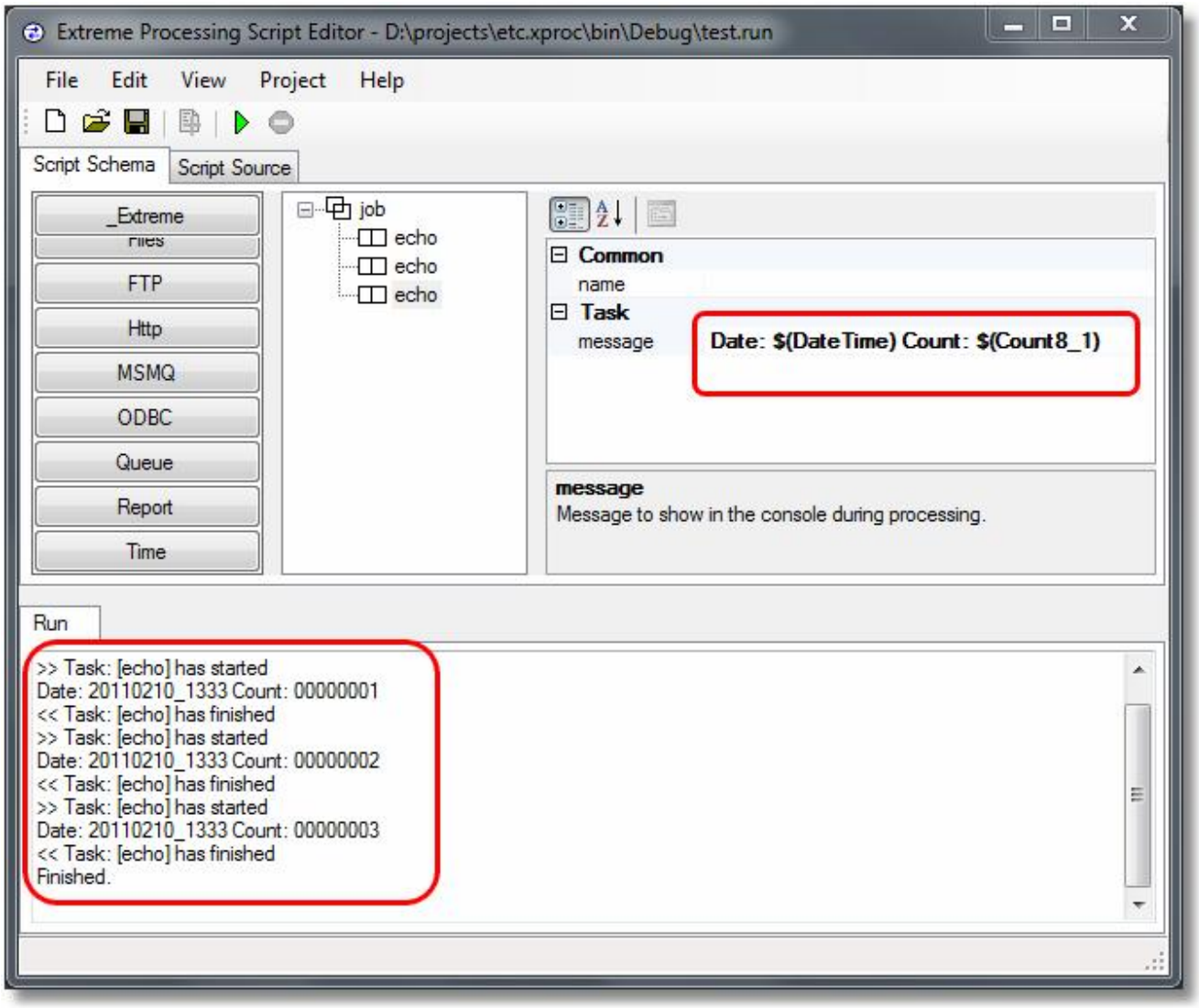


There is example of desktop shortcut that calls Script Editor. This example shows how to pass parameter into Script Editor tool in order to run script with the property “dir” that accepts parameter called $(directory).

Macros

Macros are similar to expressions covered in chapter “Passing Parameters”. They are special expressions enclosed in between “$(“ and “)”. During script execution those special expressions are replaced by dynamic values of date, date time or internal counts.
Macros are mostly used to create messages with date time embedded in them or create unique output file names during processing.
Macros are case-sensitive.

Macro	*Result*
$(DateTimeShort)	Produces date and time in format YYMMDD_hhmmss
$(DateTimeLong) or $(DateTime)	Produces date in format CCYYMMDD_hhmmss
$(DateShort)	Produces date in format YYMMDD
$(DateLong) or $(Date)	Produces date in format CCYYMMDD
$(Count1) or $(Count2) or $(Count3)	Produces internal count
$(Count6_1) or $(Count6_2) or $(Count6_3)	Produces internal count padded with zeros on the left with total size of 6 positions.
$(Count8_1) or $(Count8_2) or $(Count8_3)	Produces internal count padded with zeros on the left with total size of 8 positions.



There is simple example on macro use in output messages.

Job Properties

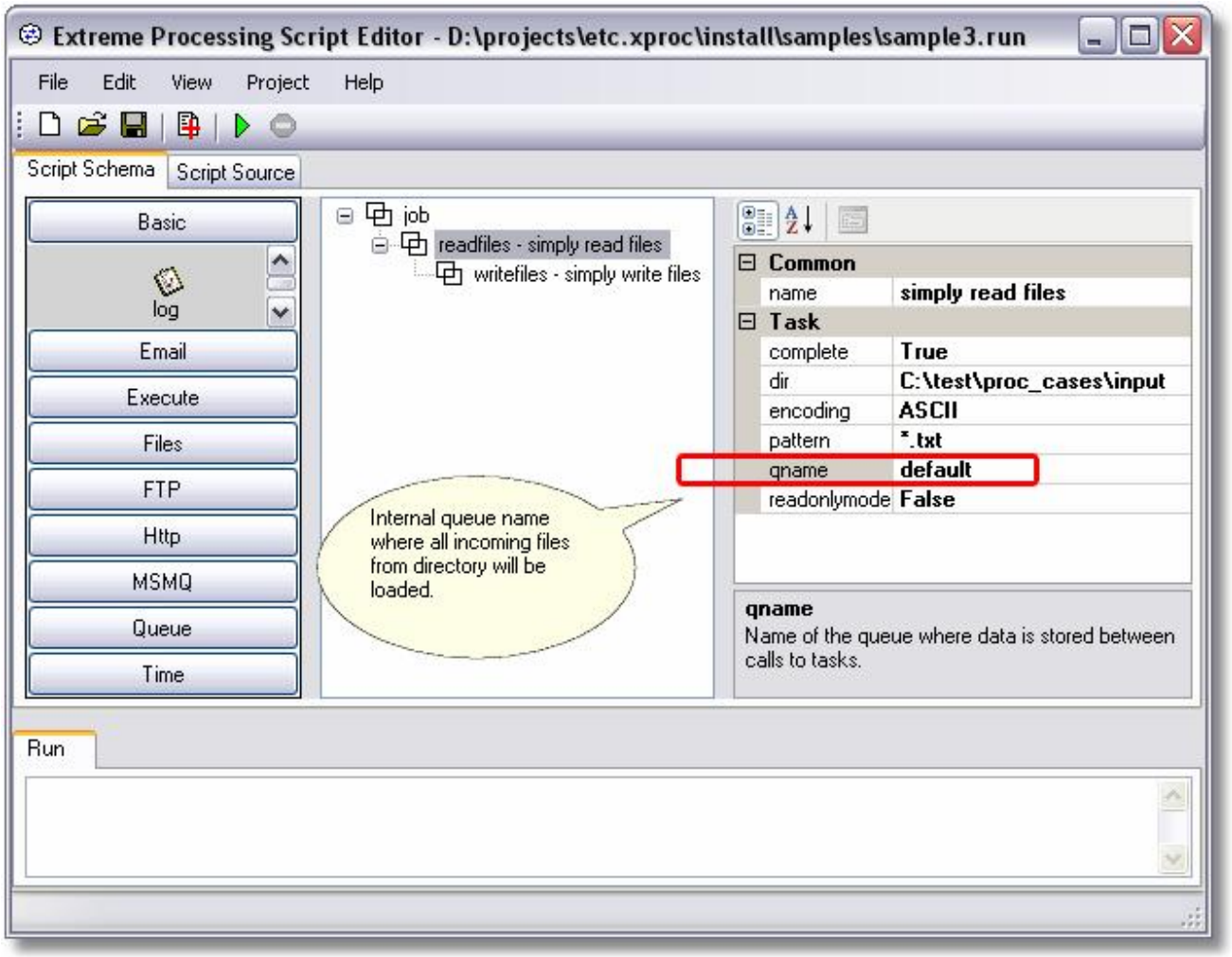
Job has number of properties that are global for whole script execution.

Property	*Description*
name	Name used to display and identify job in log file and console screen.
failonerror	If set to “false” it will try to continue job execution even if task in the job has failed.
license	License key to run this job.
tasksdir	Tasks directory defaults to “.” That means “current directory”. Tasks directory is very important as it instructs job where to look for tasks that it needs to execute.
verbose	If set to “false” it will reduce number of information messages in the output.

Internal Queue

IOTasks tasks library comes with build-in internal queue. This queue is used to keep processing data around between individual task calls. Some tasks are also created to clean or manage internal queue. IOTasks library uses queue called “default”, however they are not limited to one queue. You can have more than one internal queue by simply calling them with different “qname”. For example: you can have one queue to read email messages via POP3 and write to local files, and another queue to read files from FTP and send them using email via SMTP.

Internal queue concept is important for IOTasks tasks library to function. It allows data flow between tasks and provides support for many additional tasks to filter and process data. Many tasks in IOTasks library have “qname” property that is set to “default”. This property indicates what internal queue will be used to read data from or write data into. In most cases you will only need one queue per job to read data in, process it and write it out.



Internal queue name is set to “default” but more than one internal queue can be used for complex job scripts when multiple streams of data required. Simply use different value in qname for other streams of data if you need more than one.

Tasks

Most tasks are independent building blocks of the job. Tasks can be added to the list of available tasks by simply placing new DLL containing new tasks into job's "tasksdir". You or third parties can also develop them independently. Tasks provide an extensible framework as new tasks can be added without a need to recompile or rebuild whole application. Some basic tasks come pre-packaged with the runtime.

Build-in Tasks

Task	*Properties*	*Usage*
echo		Task sends simple information messages to the output console or notification window. It is very helpful in testing and debugging job execution.
	message	Message to be produced in the output screen or command line window. This task is very useful for diagnostic and testing purposes.
pause		Task pauses job execution for predefined time
	minutes	Number of minutes to wait
	seconds	Number of seconds to wait
interval		Task runs other tasks nested within itself at predefined time intervals.
	minutes	Number of minutes to wait before executing tasks nested in it.
	seconds	Number of seconds to wait before executing tasks nested in it.
time		Task runs other tasks nested within itself at predefined times everyday.
	schedule	Time schedule with hours and minutes separated by comma. Example: 10:30AM,11:45AM,1:20PM this schedule will trigger this task at these predefined times. If specific time already passed today, it will trigger again tomorrow.
calendar		Task runs other tasks nested within itself at predefined date and time.
	schedule	Datetime schedule with date, hours and minutes separated by comma. Example: 1/14/2006 10:20AM,12/25/2006 3:20PM this schedule will trigger this task at these predefined dates. If date is in the past, it will trigger again after 1 year. This way you can setup schedule to run tasks at specific date every year.
subjob		Sub job is an independent execution unit inside of the job. Sub job starts new thread and runs tasks nested inside. It is recommended to avoid using Sub job as it may complicate your script execution. Use Sub jobs only if you have no other solution.
	N/A	
property		Reserved for future use.
	key	Unique key to identify property.
	value	Value for the property. Property task is used to define constants in scripts.
loop		Execute nested tasks in the infinite loop. This task can be used for some service jobs running in the background all the time.
	N/A	
log		Log notification messages into the file using this task.
	file	Log file name with directory.
	message	Information message to be written to the log file.
	timestamp	If set to "true" will write current timestamp into the output log file.
logfilenames		Log file names currently in the queue into the file using this task.

	file	Log file name with directory.
	message	Information message to be written to the log file.
	timestamp	If set to “true” will write current timestamp into the output log file.
fail		Task throws exception in order to abort job execution.
	message	Message to be produced in the output just before job fails. This task is used to abort job execution, or force job execution out of looping.
catch		Catch errors in nested tasks. This task is useful if you want to proceed with job execution even if errors occurred in the tasks nested inside <catch> task. Catch task can be very useful for tasks used in data communications. Example: let say if <readftp> task fails to connect to FTP server, you want to try reconnect after 10 minutes; in this case catch will absorb <readftp> failure and keep job executing.
	N/A	
exit		Exit job gracefully.
	N/A	
exec		Execute external program and wait for its completion.
	program	External application or program to run. Example: C:\Windows\notepad.exe
	parameters	Parameters to pass to external program.
	workingdir	Working directory.
	timeout	Number of seconds to wait until program finishes.
	checkexitcode	If set to true, will check exit code from external program.
	ifexitcode	If exit code returned by the application matches one set in this property, then execute tasks nested in this task. Check is performed only if checkexitcode property is set to true.
execjob		Execute external job script.
	file	File name of the external job.
	parameters	Parameters to pass to external job.
onerror		Execute external job script and if it fails then run nested tasks. This is like a combination of <execjob> and <catch> tasks in one task.
	file	File name of the external job.
	parameters	Parameters to pass to external job.
GC		Run garbage collector (GC). This task is used for long running intense scripts that process data in real-time or near real-time. Use this task to recycle memory used by the script.

Additional Input/Output Tasks (IOTasks)

Task	*Parameters*	*Usage*
readfiles		Task will read files into internal queue based on the directory and file pattern.
	qname	Internal queue where files should be placed.
	dir	Directory where files are.
	pattern	Only file names matching this pattern should be read. Examples of patterns: *.txt will read all text files from specified directory.
	encoding	Encoding to use when reading the files. Possible values: ASCII, European_8bit, Unicode.
	complete	This option slows down reading task significantly if there are many files to read. If set to true will check if file is complete before reading it. If other

		process or program is writing same file at the same moment, this option will attempt to take file only when other process is done with writing it. This option cannot guarantee the file is complete if other process's write is slow or other process is appending to the file constantly.
	readonlymode	If set to true will read files but keep original files and do not remove them from source directory after read.
writefiles		Task will write files from internal queue into directory using defined naming pattern.
	qname	Internal queue where files should be written off.
	dir	Directory where files will be placed.
	pattern	File name or pattern to write multiple files. Macros can be inserted in order to produce unique file names or file names that match input files. Available macros are: %count% - produce unique number in the file name %date% - produce date in the file name %time% - produce time in the file name %source% - produce input file or source name.
	encoding	Encoding to use when reading the files. Possible values: ASCII, European_8bit, Unicode.
	append	If set to true will append to existing files instead of overwrite.
filewatch		Task will monitor directory based on file name pattern.
	dir	Directory where files should be monitored
	pattern	Only file names matching this pattern should be watched. Examples of patterns: *.txt will monitor all text files from specified directory.
	changeevent	If set true, will monitor for file change events
	createevent	If set true, will monitor for file create events
	renameevent	If set true, will monitor for file rename events
	deleteevent	If set true, will monitor for file delete events
deletefiles		Task will delete files in the directory based on file name pattern.
	dir	Directory where files are.
	pattern	Only file names matching this pattern should be deleted. Examples of patterns: *.txt will delete all text files from specified directory.
movefiles		Task will move files from source to destination directory based on file pattern.
	sourcedir	Directory where files are.
	destinationdir	Directory where files will be moved.
	pattern	Only file names matching this pattern should be moved. Examples of patterns: *.txt will delete all text files from specified directory.
	overwrite	If set to true will overwrite existing destination files.
	copy	If set to true will copy files instead of moving them.
movefilesonebyone		Task will move files from source to destination directory based on file pattern. It also runs other nested tasks below on every copied/moved file.
	sourcedir	Directory where files are.
	destinationdir	Directory where files will be moved.
	pattern	Only file names matching this pattern should be deleted. Examples of patterns: *.txt will delete all text files from specified directory.
	overwrite	If set to true will overwrite existing destination files.
	copy	If set to true will copy files instead of moving them.
splitfiles		Task will split files in the directory based on certain text inside those files. Example: if you set boundary to ST* it will split incoming EDI X12 files into separate transactions (separate ST/SE blocks). See <splitq> task that

		also splits files based on data in them. <splitq> works on the queue while <splitfiles> works on files.
	dir	Directory where files are.
	pattern	Only file names matching this pattern should be split. Examples of patterns: *.txt will split all text files from specified directory.
	outputpattern	Pattern for output file naming. Example: out*.txt will create files out1.txt, out2.txt, out3.txt, etc.
	boundary	Any text that will indicate beginning of the new file. Example: EDI X12 Purchase Orders 850 could be split using boundary set to ST* or ST*850.
	keepheader	If set to true will keep header data and produce all the split files with it at the top of the file. Example: if you split EDI X12 files using ST* your header will be ISA, GS segments of the original file.
iffilesexist		Task will check if files matching the pattern exist in the directory, if files exist then nested tasks will be executed.
	dir	Directory where to search for files. This task will execute nested tasks only if at least one file is found that matches include/exclude patterns.
	include	Include files that match this pattern. Pattern examples: *.txt or *.xml.
	exclude	Exclude files that match this pattern. Pattern examples: *.txt or *.xml.
readftp		Task will read files from FTP server directory into internal queue.
	qname	Internal queue where FTP files will be placed.
	server	FTP server name.
	port	FTP server port.
	dir	FTP server directory.
	pattern	Remote file name pattern to select multiple files. Example: *.txt would match all the text files in remote directory.
	username	FTP server user name.
	password	User password to log into FTP server.
	binary	If set to true will use binary data transfer mode.
	readonlymode	If set to true will read files but keep original files on server and do not remove them from remote directory after read.
	passive	If set to true will use passive connection mode.
	keepalive	If set to true will keep connection open between FTP requests.
	ssl	If set to true will use FTP over SSL (FTPS) channel.
	certificatefile	X.509 certificate file will be loaded using file name with full path set in this property. Works only if ssl property is also set to true.
	timeout	Communication timeout value in number of seconds.
	standardftp	If set to true will use standard FTP, provide support for SSL, passive mode, keep alive and all other properties, if set to false will use only small subset of FTP features to allow communication with old or not fully RFC compatible systems.
writeftp		Task will write files from internal queue into FTP server directory.
	qname	Internal queue where files are before they are written to FTP server.
	server	FTP server name.
	port	FTP server port.
	dir	FTP server directory.
	pattern	Remote file name or pattern to write multiple files. Macros can be inserted in order to produce unique file names or file name that match input files. Available macros are: %count% - produce unique number in the file name %date% - produce date in the file name %time% - produce time in the file name %source% - produce input file or source name.
	username	FTP server user name.
	password	User password to log into FTP server.

	binary	If set to true will use binary data transfer mode.
	passive	If set to true will use passive connection mode.
	keepalive	If set to true will keep connection open between FTP requests.
	ssl	If set to true will use FTP over SSL (FTPS) channel.
	certificatefile	X.509 certificate file will be loaded using file name with full path set in this property. Works only if ssl property is also set to true.
	timeout	Communication timeout value in number of seconds.
	standardftp	If set to true will use standard FTP, provide support for SSL, passive mode, keep alive and all other properties, if set to false will use only small subset of FTP features to allow communication with old or not fully RFC compatible systems.
ifftpfilesexist		Task will check if FTP files matching pattern exist in server directory, if files exist then tasks nested in this task will be executed.
	server	FTP server name.
	port	FTP server port.
	dir	FTP server directory.
	pattern	Remote file name or pattern to select multiple files.
	username	FTP server user name.
	password	User password to log into FTP server.
	passive	If set to true will use passive connection mode.
	keepalive	If set to true will keep connection open between FTP requests.
	ssl	If set to true will use FTP over SSL (FTPS) channel.
	certificatefile	X.509 certificate file will be loaded using file name with full path set in this property. Works only if ssl property is also set to true.
	timeout	Communication timeout value in number of seconds.
readpop3		Task will read messages from email server into internal queue.
	qname	
	server	POP3 email server.
	port	POP3 port.
	username	User name used to read email.
	password	User password for POP3 server.
	attachmentsonly	Messages are in attachments. Ignore email body and do not create message for it in the internal queue.
	subjectcontains	Read and add message to internal queue only if email subject contains text defined in this property.
	readonlymode	Keep copy of the email on server, do not remove email message after it was read. This option assumes that some other process will remove email messages from email server.
writesmtp		Task will write messages from internal queue into email server.
	qname	Internal queue that will provide messages for the SMTP email server.
	server	SMTP email server name.
	port	SMTP port number.
	from	From email address.
	to	To email address.
	subject	Email subject line.
	body	Email body.
	pattern	Email attachment file name or pattern to name multiple files. Macros can be inserted in order to produce unique file names or file names that match input files. Available macros are: %count% - produce unique number in the file name %date% - produce date in the file name %time% - produce time in the file name %source% - produce input file or source name.
	useattachments	If set to true will use attachments instead of body to send messages.

notifysmtp		Task will send simple email notification message. This task can be used to notify system administrators about job execution and status.
	server	SMTP email server name.
	port	SMTP port number.
	from	From email address.
	to	To email address.
	subject	Email subject line.
	body	Email body.
onerrorsmtp		Task will send simple email notification message when any of the tasks nested in it fail. This task can be used to notify system administrators about job execution and status.
	server	SMTP email server name.
	port	SMTP port number.
	from	From email address.
	to	To email address.
	subject	Email subject line.
	body	Email body.
readhttp		Read messages via internet and place them into internal queue.
	qname	Internal queue name where to place messages received via HTTP.
	server	Web server name.
	port	Web server port number.
	binary	If set to “true” will use binary transfer mode.
writehttp		Read messages from internal queue and post them to web server.
	qname	Internal queue name where to place messages received via HTTP.
	server	Web server name.
	contenttype	Default value application/x-www-form-urlencoded for posting HTML.
	method	Default value is POST for posting to web server.
readmsmq		Read messages from MSMQ and place them into internal queue for future processing.
	qname	Internal queue name where messages will be taken to.
	msmqname	MSMQ name. Example: .\private$\myQueue
	seconds	Number of seconds to wait for every incoming message.
writemsmq		Write messages from internal queue into MSMQ.
	qname	Internal queue name where messages will be taken from.
	msmqname	MSMQ name. Example: .\myQueue
ifqnotempty		Task will execute other tasks nested in itself if internal queue is not empty.
	qname	If queue has messages, execute nested tasks.
ifqempty		Task will execute other tasks nested in itself if internal queue is empty.
	qname	If queue has no messages, execute nested tasks.
clearq		Task will clear internal queue.
	qname	Clear internal queue from all messages it contains.
filterq		Task will filter and remove messages from internal queue that match specific criterion.
	qname	Internal queue name.
	contains (see “to” and “from”)	Filter message from queue if message contains this text.
	to	Start looking for “contains” text up to this text.

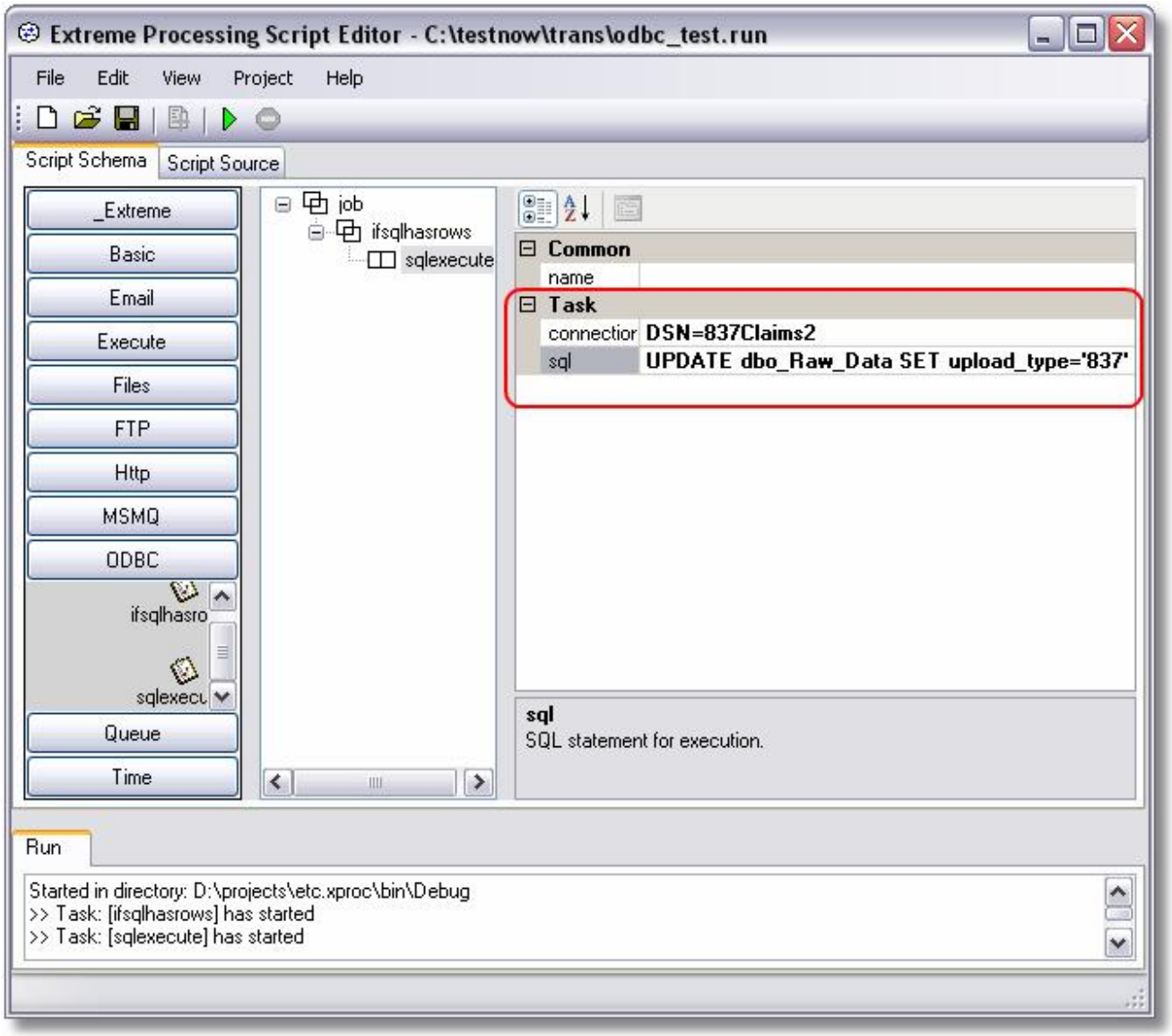
	from	Start looking for “contains” text from this text.
	startswith	Filter message from queue if message starts with this text.
	endswith	Filter message from queue if message ends with this text.
keepq		Task will keep messages in the internal queue only if they match specific criterion. Essentially this task is an opposite to “filterq” task.
	qname	Internal queue name.
	contains (see “to” and “from”)	Keep message in queue if message contains this text.
	to	Start looking for “contains” text up to this text.
	from	Start looking for “contains” text from this text.
	startswith	Keep message in queue if message starts with this text.
	endswith	Keep message in queue if message ends with this text.
splitq		Task will split messages in the internal queue based on structure of the data in the message. Essentially this task is pre-processing task to “filterq” or “keepq” task. It is used to split big EDI X12 and EDIFACT files into smaller interchanges and separate messages.
	qname	Internal queue name.
	x12_interchange	If set to true split EDI X12 block of data into separate X12 interchanges.
	edifact_message	If set to true split EDIFACT block of data into separate EDIFACT messages.
	pattern	File name or pattern to form multiple files. Macros can be inserted in order to produce unique file names or file names that match input files. Available macros are: %count% - produce unique number in the file name %date% - produce date in the file name %time% - produce time in the file name %source% - produce input file or source name.
qproperties		Reserved for future use.
	persistent	If set to true will persist each queue message in internal storage for safety. If job fails or computer crashes messages can be recovered.

Additional SQL Database Tasks (ODBCTasks)

ODBC tasks use odbctasks.dll library. You will need to setup ODBC data source in Control Panel to use those tasks. You can perform simple database maintenance using ODBC tasks: use <ifsqlhasrows> task to check for existing records, and then perform actions using <sqlexecute>.

Task	*Parameters*	*Usage*
sqlexecute		Task will execute SQL query using ODBC connection.
	sql	SQL statement in a form of INSERT, UPDATE or DELETE.
	connection	Connection string to ODBC data source. Example: DSN=test;UID=myname;PWD=mypassword
ifsqlhasrows		Task will execute SQL query using ODBC connection. If resulting record set contains any rows, nested tasks will be executed. This task could be used in combination with <sqlexecute> task.
	sql	SQL statement in a form of SELECT.
	connection	Connection string to ODBC data source. Example: DSN=test;UID=myname;PWD=mypassword
untilsqlhasrows		Task will execute SQL query using ODBC connection. If resulting record set contains any rows, nested tasks will be executed. This task could be used in combination with <sqlexecute> task. Task will loop (repeat) until SQL returns

		any rows that's why it is important for nested tasks to update database otherwise this task will loop indefinitely.
	sql	SQL statement in a form of SELECT.
	connection	Connection string to ODBC data source. Example: DSN=test;UID=myname;PWD=mypassword



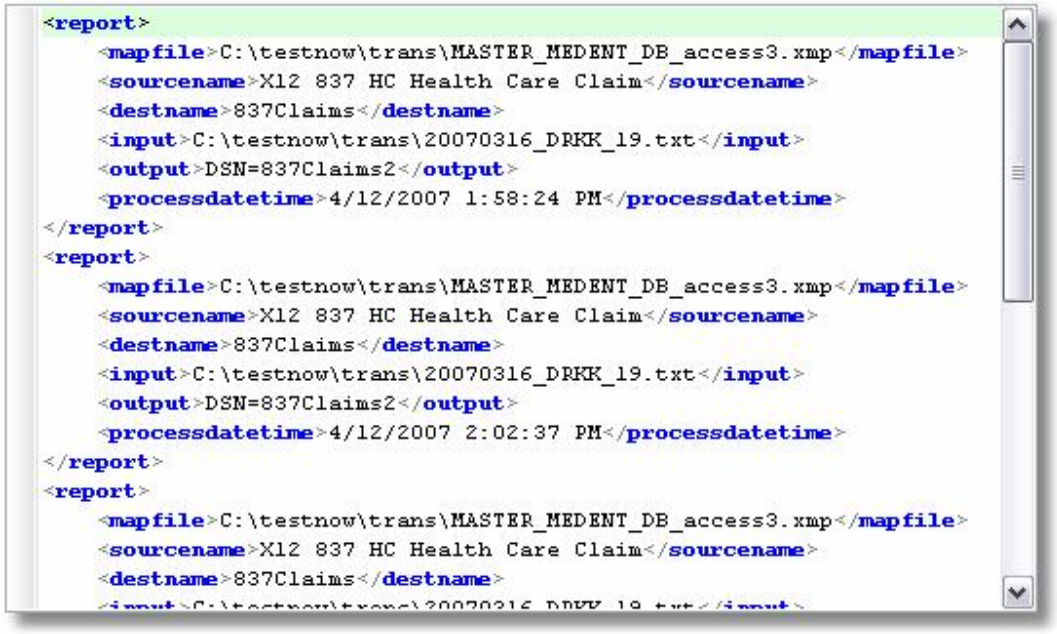
Example of ODBC tasks in action.

Additional Reporting Tasks (RPTTasks)

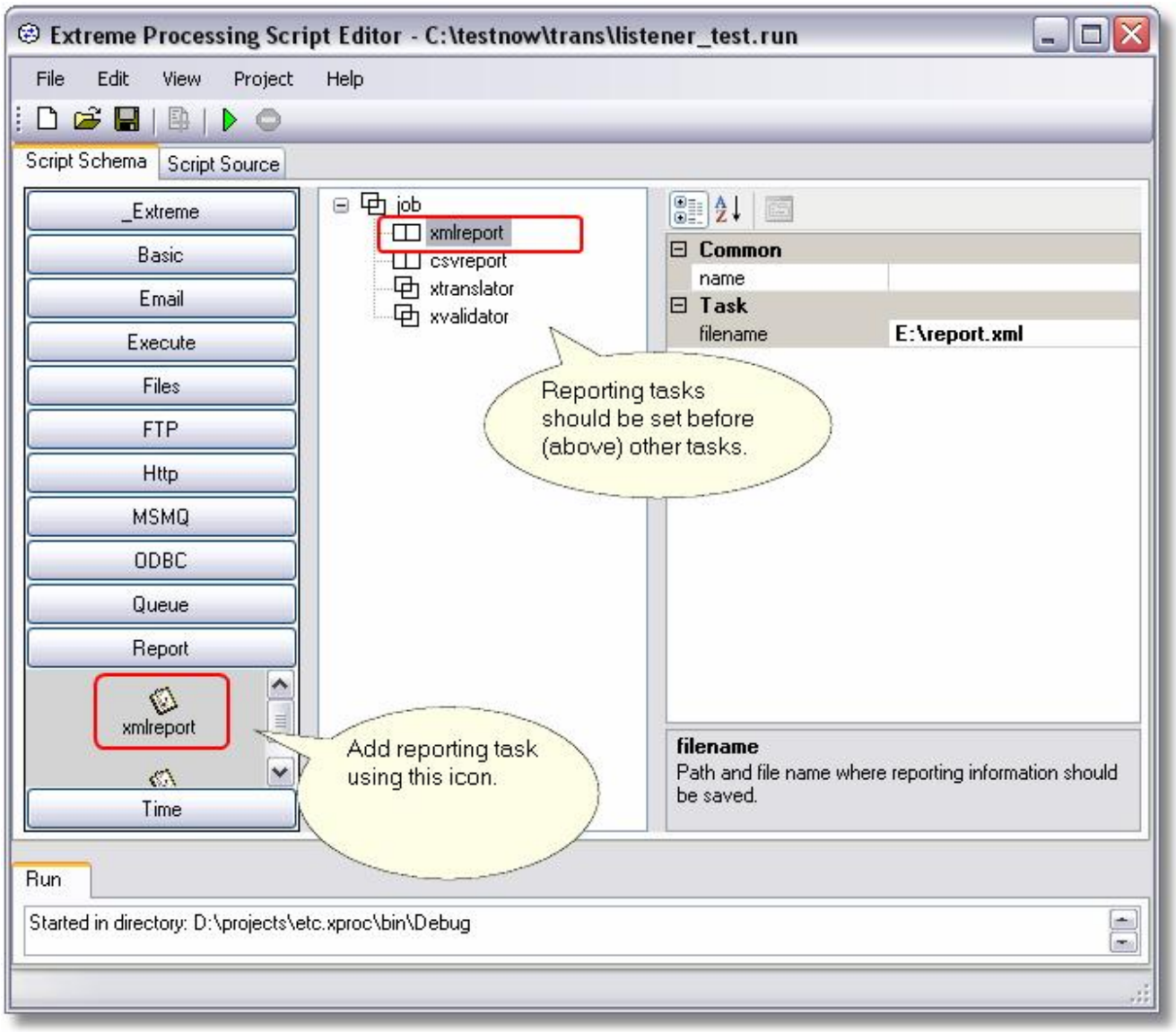
Reporting tasks use rpttasks.dll and epreport.dll libraries. Reporting tasks should be set at the top of job execution tree, above most of other tasks. Reporting tasks capture and save reporting information from <xtranslator>, <xvalidator> and other tasks.

Task	*Parameters*	*Usage*
csvreport		Task will prepare processing reports in CSV format based on results from other tasks such as <xtranslator> or <xvalidator> and save them into CSV file.
	filename	Path and file name where report should be saved in CSV format (Comma

		Separated Value flat file).
xmlreport		Task will prepare processing reports in XML format based on results from other tasks such as <xtranslator> or <xvalidator> and save them into XML file.
	filename	Path and file name where report should be saved in XML format.
sqlreport		Task will prepare processing reports based on results from other tasks such as <xtranslator> or <xvalidator> and save them into SQL database via ODBC connection.
	connection	Connection string to ODBC data source. Example: DSN=test;UID=myname;PWD=mypassword
	table	Table name in the database that will hold saved reports. Table must be pre-created before reporting begins. Task can not create table if it does not exists. Task only inserts new reports it can not update them. Default suggested table name is EP_REPORT. If you choose other name, do not use any spaces in the name.

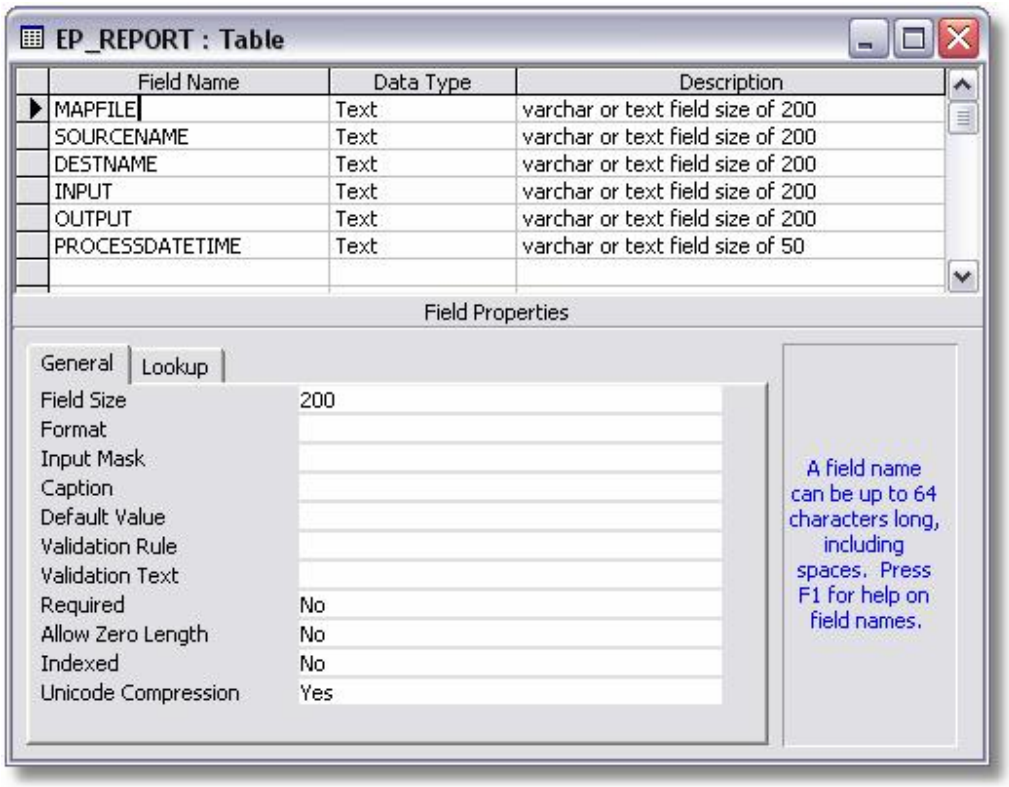


<xmlreport> reporting task saves processing results in XML format.



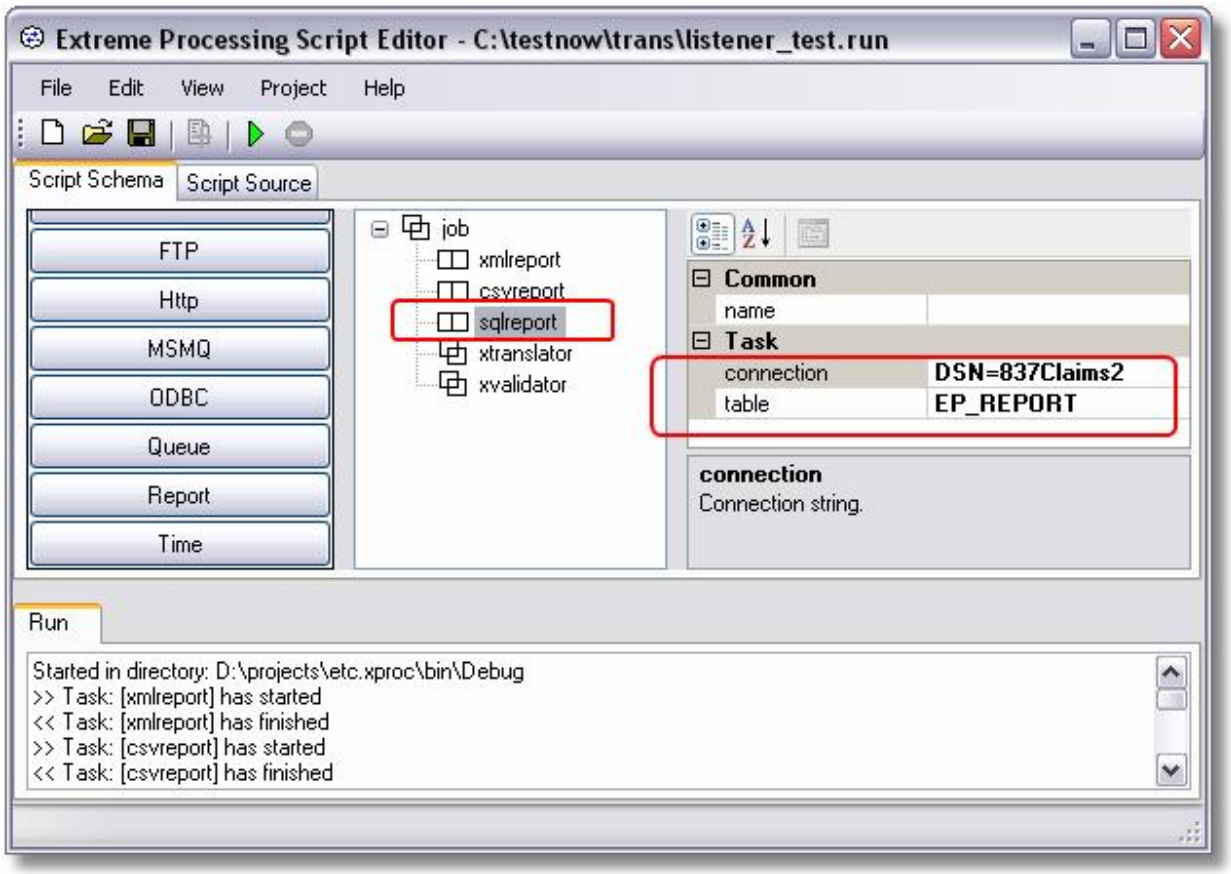
Example of reporting task setup.

You can use SQL reporting task to save reports into SQL database via ODBC. <sqlreport> task needs database connection information and table name where report data will be logged.



There is layout of the MS Access table. You can use almost any SQL based database that you can connect to using ODBC data source defined in the Control Panel.

This is a view of the table that contains report logged with <sqlreport> task.



<sqlreport> task is used to log data into the SQL database via ODBC connection. <sqlreport> task should be set before (above) <xtranslator> and <xvalidator> tasks in the Extreme Processing screen.

Additional Translator Tasks (TRTasks)

xtranslator task uses xtraneng.dll library from Extreme Translator software package. If you use xtranslator task in your script make sure that xtraneng.dll is the same as one used by Extreme Translator installation. You can copy over xtraneng.dll from Extreme Translator installation to make sure your map execution uses same runtime version.

Task	*Parameters*	*Usage*
xtranslator		Task will run Extreme Translator map.
	mapfile	Extreme Translator map file (including path).
	license	Extreme Translator license key for the map to run.
	input	Input file name and directory. Word “Input” (without quotes) has to be entered in DataPath property for the input side of the map.
	output	Output file name and directory. Word “Output” (without quotes) has to be entered in DataPath property for the output side of the map.
	parameters	Additional parameters separated by spaces.
	includedirectories	Include subdirectories when processing input files. If set to true, looks for files not just in current directory set but “input” property but also looks for files in all subdirectories below current.

Setup procedures:
1. In the translator Map Editor you need to enter word Input in the input side DataPath and word Output in the output side DataPath.

2. Once you do that, Extreme Processing will be able to pass values into your map. Without words Input and Output in DataPath properties, map will use whatever values are set in the map already and values you set in the Extreme Processing will not get passed to the map.

3. Now once you set DataPath to words Input and Output. You will notice that you can not run map in the Map Editor. You can still do that by going to "Runtime Parameters.. " in the Map Editor and setting those values in that screen. After that you should be able to run map in the Map Editor and also use it in the Extreme Processing.

Additional FormXT Tasks (FXTasks)

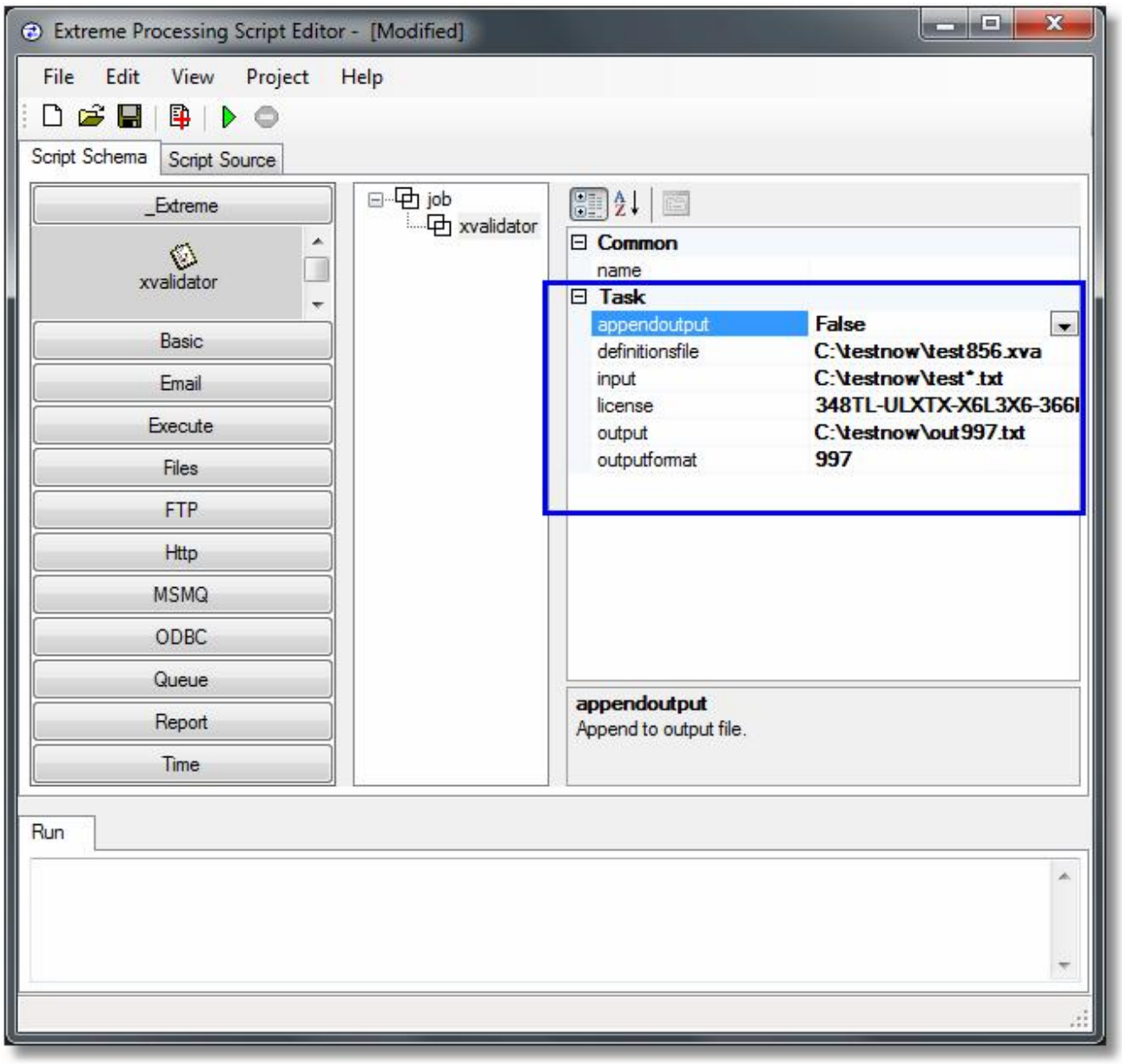
formxt task uses formxteng.dll and found.dll libraries from FormXT software package. If you use formxt task in your script make sure that formxteng.dll and found.dll is the same as one used by FormXT installation. You can copy over formxteng.dll and found.dll from FormXT installation to make sure your execution uses same runtime version.

Task	*Parameters*	*Usage*
formxt		Task will run FormXT map.
	mapfile	FormXT map file (including path).
	license	FormXT license key for the map to run.
	input	Input file name and directory.
	output	Output file name and directory.
	parameters	Additional parameters separated by spaces.

Additional Validator Tasks (VATasks)

xvalidator task uses valideng.dll library from EDI Validator software package. If you use xvalidator task in your script make sure that valideng.dll is the same as one used by EDI Validator installation. You can copy over valideng.dll from EDI Validator installation to make sure your validation execution uses same runtime version.

Task	*Parameters*	*Usage*
xvalidator		Task will run EDI Validator schema definitions file.
	appendoutput	If set to True appends to output file instead of overwrite. If set to False use macros such as %Count% to create unique output file names.
	definitionsfile	Extreme Validator definitions file (including path).
	license	Extreme Validator license key for the map to run.
	input	Input file name. Input file name contains data that will be validated. Example: C:\test\somedirectory\test*.txt This will pickup all text files that start with word “test” in specific directory.
	output	Output file name. Output file will contain validation results. Example: C:\test\somedirectory\test_out.txt
	outputformat	Output file format. Accepted values: XML or Plain.



Example of xvalidator task setup.

Container Task Execution

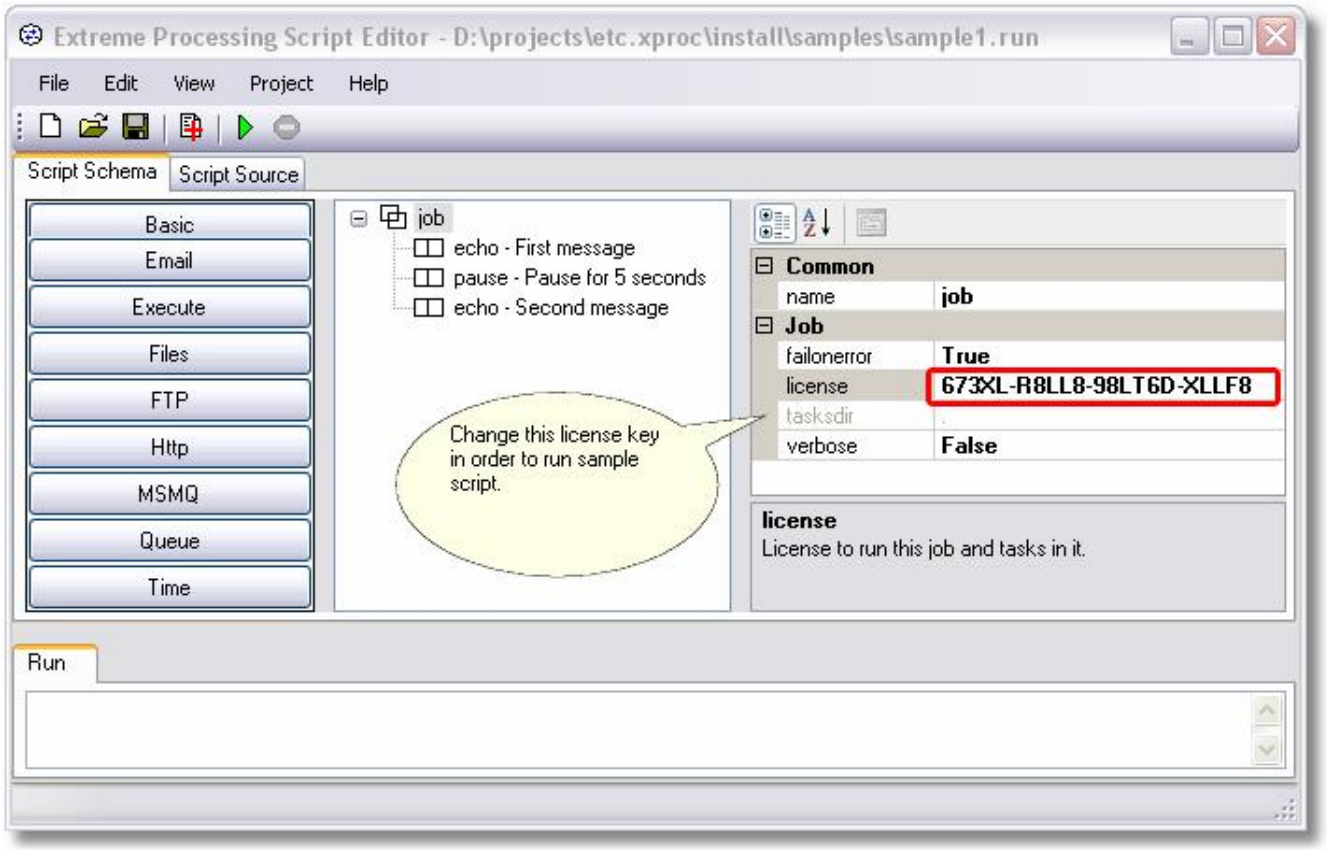
Tasks nested in other container tasks are executed if following conditions are met during container execution

Container Task	*Condition under which it will execute nested tasks*
interval	When interval defined in minutes and seconds will pass.
time	When time defined in schedule will come.
calendar	When date and time defined in schedule will come.
subjob	Will always execute tasks nested in it, but run them in separate thread.
loop	Will always execute tasks nested in it, but run them repeatedly in the loop.
catch	Will always execute tasks nested in it, but suppress any errors that might happen during execution of those tasks nested.
exec	If external program runs and returns expected exit code (if specific exit code is expected)
execjob	If external job runs and does not fail.

readfiles	If files have been read from specified directory.
writefiles	If files have been written to specified directory.
iffilesexist	If files exist in specific directory.
readftp	If files have been read from FTP server remote directory.
writeftp	If files have been written to FTP server remote directory.
ifftpfilesexist	If files exist on FTP server remote directory.
readpop3	If email messages have been read from the POP3 mail server.
writesmtp	If email messages have been sent to SMTP mail server.
readhttp	If file has been read from HTTP Web server.
writehttp	If file has been written to HTTP Web server.
readmsmq	If messages have been read from MSMQ.
writemsmq	If messages have been written to MSMQ.
ifqnotempty	If internal queue is not empty and contains messages.
ifqempty	If internal queue is empty and contains no messages.
ifsqlhasrows	If resulting SELECT statement record set contains any rows.
xtranslator	If Extreme Translator map runs without errors.
formxt	If FormXT map runs without errors.
xvalidator	If Extreme Validator validation runs without errors.

Samples

Samples are located in \samples directory under product installation directory. They show basic use of the product. In order to run samples you might need to change license key that comes with the sample script you want to run.

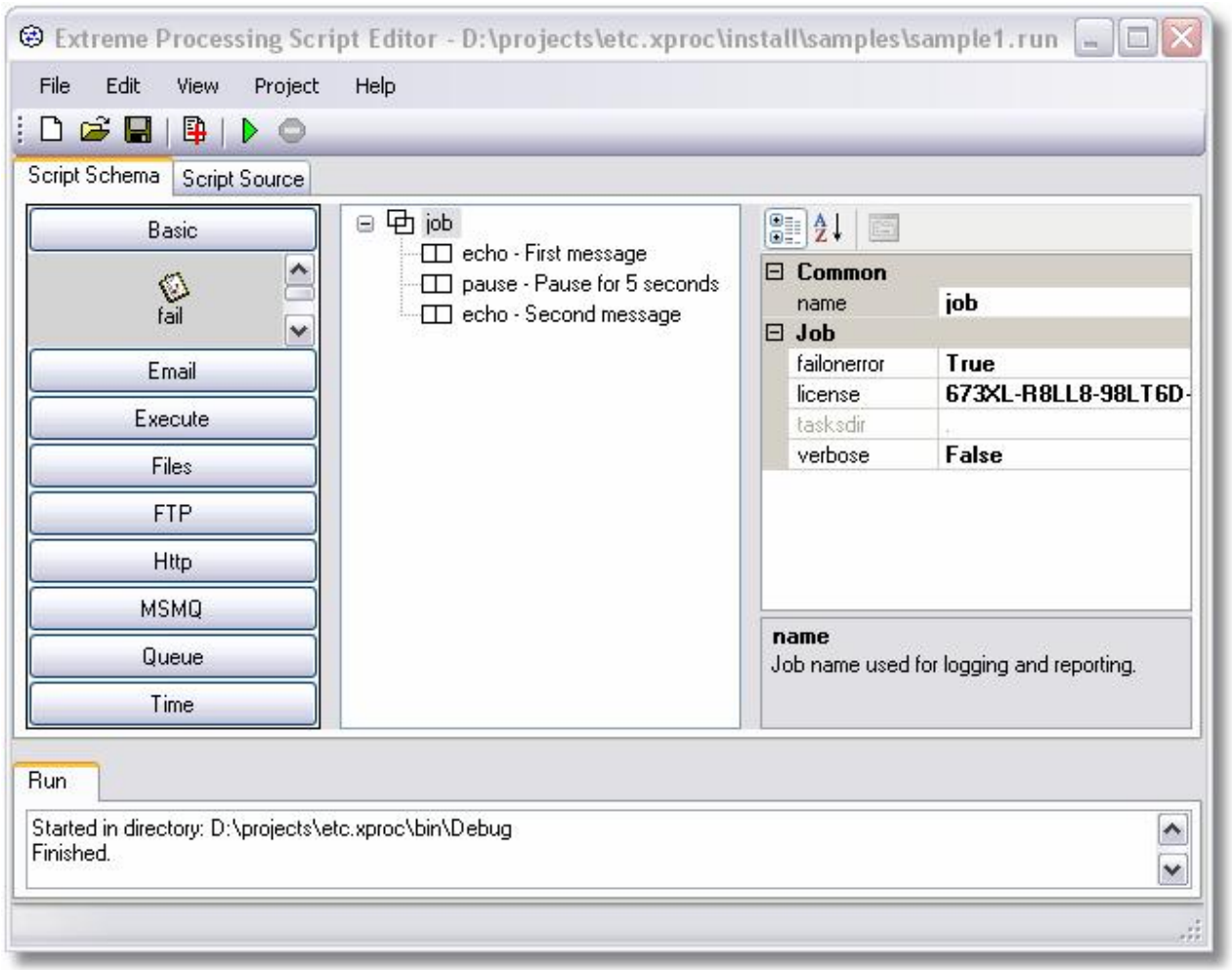


Software should use your license key to run samples, if sample fails to run try to copy your license key from Product License dialog screen into the sample script you want to run.

Every script can be opened in XML capable editor and edited or viewed. Tasks in scripts are usually indented if they are contained by other tasks.

```
<job name="job" license="demo" failonerror="true" verbose="false" tasksdir=".">
    <time name="Run at specific times" schedule="5:40PM,5:42PM">
        <echo name="Show message when executed" message="Hello World!" />
    </time>
</job>
```

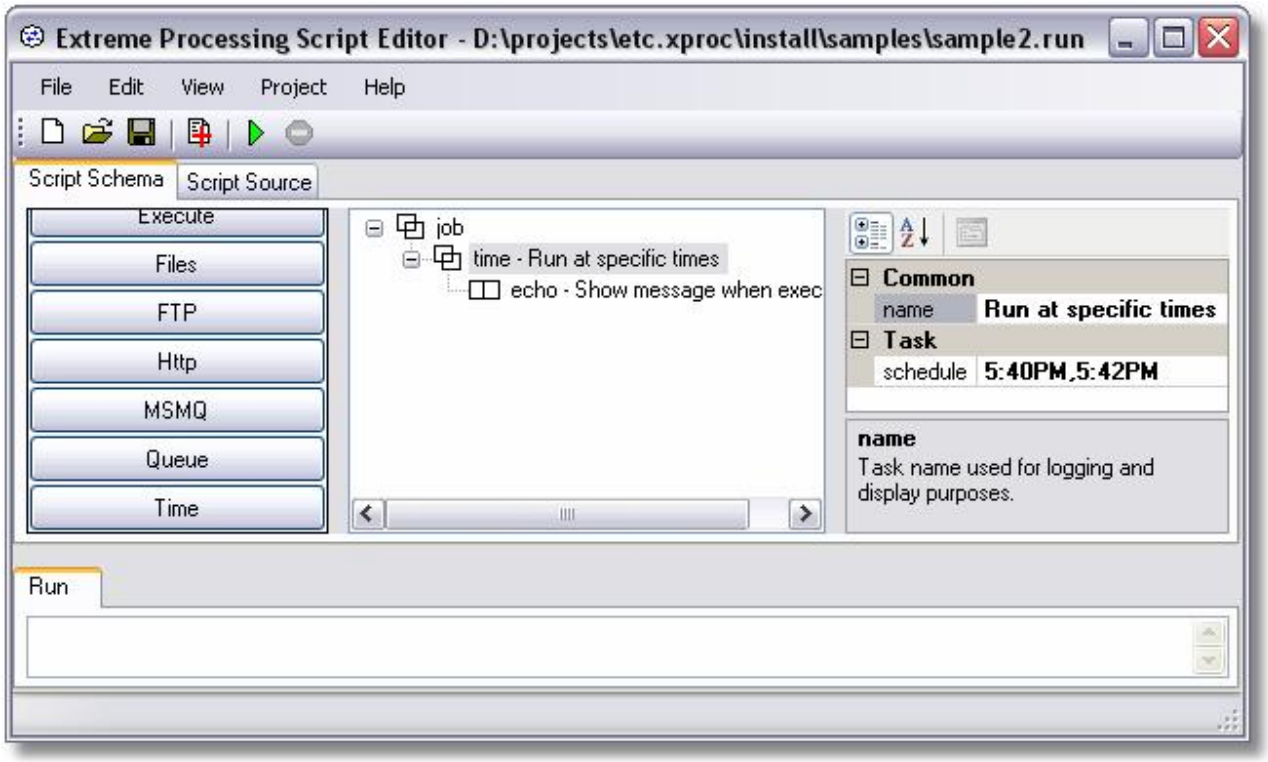
This is screen shot of sample2.run opened in XML editor. Script contains job that has a task executed at predefined times during the day.



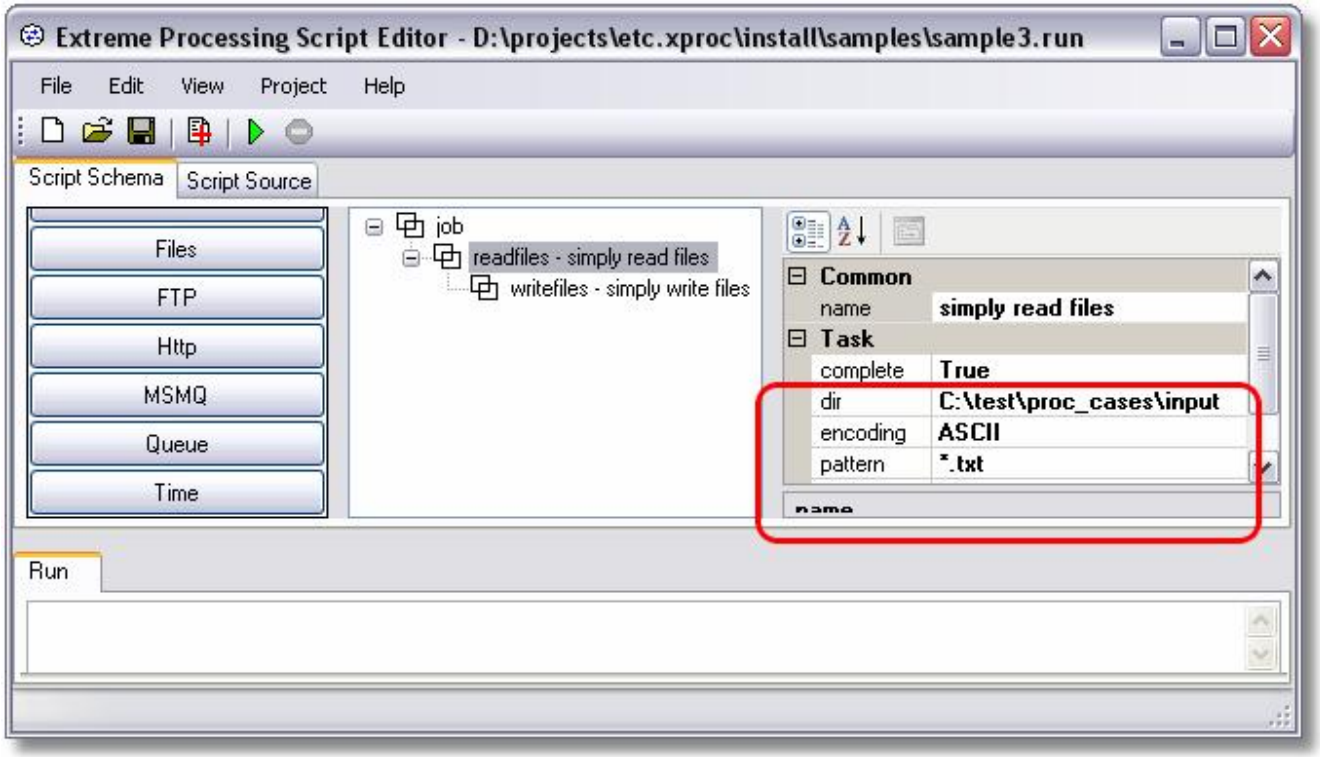
sample1.run is very basic sample script that just prints a few messages and pauses for 5 seconds.

```
<job name="job" license="demo" failonerror="true" verbose="false" tasksdir=".">
    <echo name="First message" message="Hello World!" />
    <pause name="Pause for 5 seconds" minutes="0" seconds="5" />
    <echo name="Second message" message="This is second message after the pause" />
</job>
```

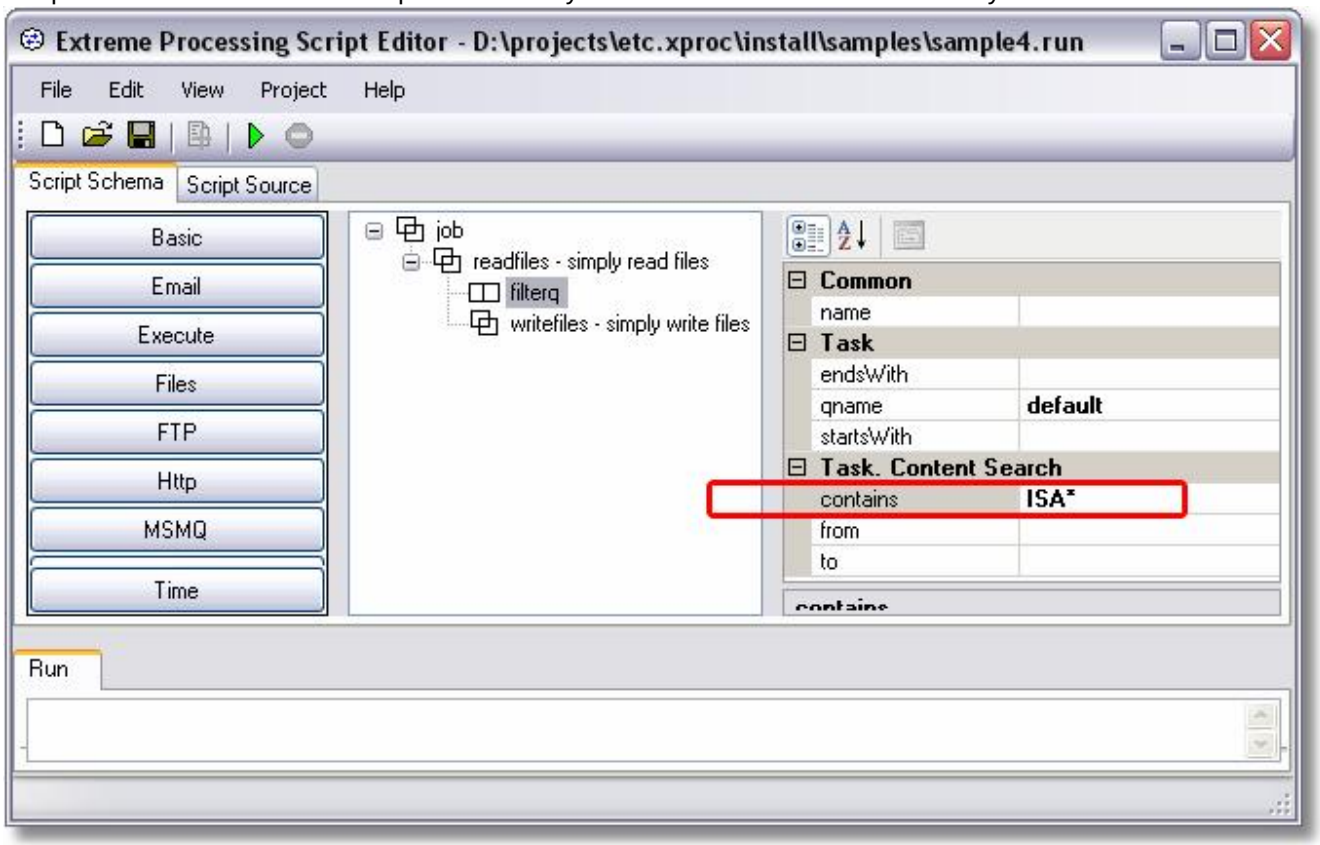
This is same sample1.run opened in XML editor.



sample2.run runs in the time loop and fires at specific times.



sample3.run reads text files from specific directory and writes them to another directory.



sample4.run is the same as sample3.run with the filter added in order to remove all the files that contain text “ISA*” in them.

Developer SDK

Developer SDK provides basic samples on how to call script jobs from .NET applications. You can call jobs directly and handle exceptions in your applications.
Source code is provided in "\DeveloperSDK" directory under product main installation directory.

```
using System;
using System.IO;

using etc.xproc;
```

New "using" clause must be included at the top of your file that will call job script. Reference to xproc.dll should be added to your project in order to have access to viJob and viJobFile classes.

```
// following code below is the most important part of this program. It uses classes from
//  etc.xproc namespace located in xproc.dll.
try
{
    viJob job = viJobFile.Load(jobFile, true, args);
    // if nothing is assigned to job.EchoHandler it will not report processing messages
    job.EchoHandler = new viConsoleEchoHandler();
    job.Execute();
}
catch (Exception exp)
{
    Console.WriteLine();
    Console.WriteLine("ERROR: " + exp.Message);
    if (exp.InnerException != null)
        Console.WriteLine("Caused by: " + exp.InnerException.Message);

    Console.WriteLine("-------- StackTrace: " + exp.StackTrace);
    if (exp.InnerException != null)
        Console.WriteLine("Caused by StackTrace: " + exp.InnerException.StackTrace);
```

This is basic job script execution example.

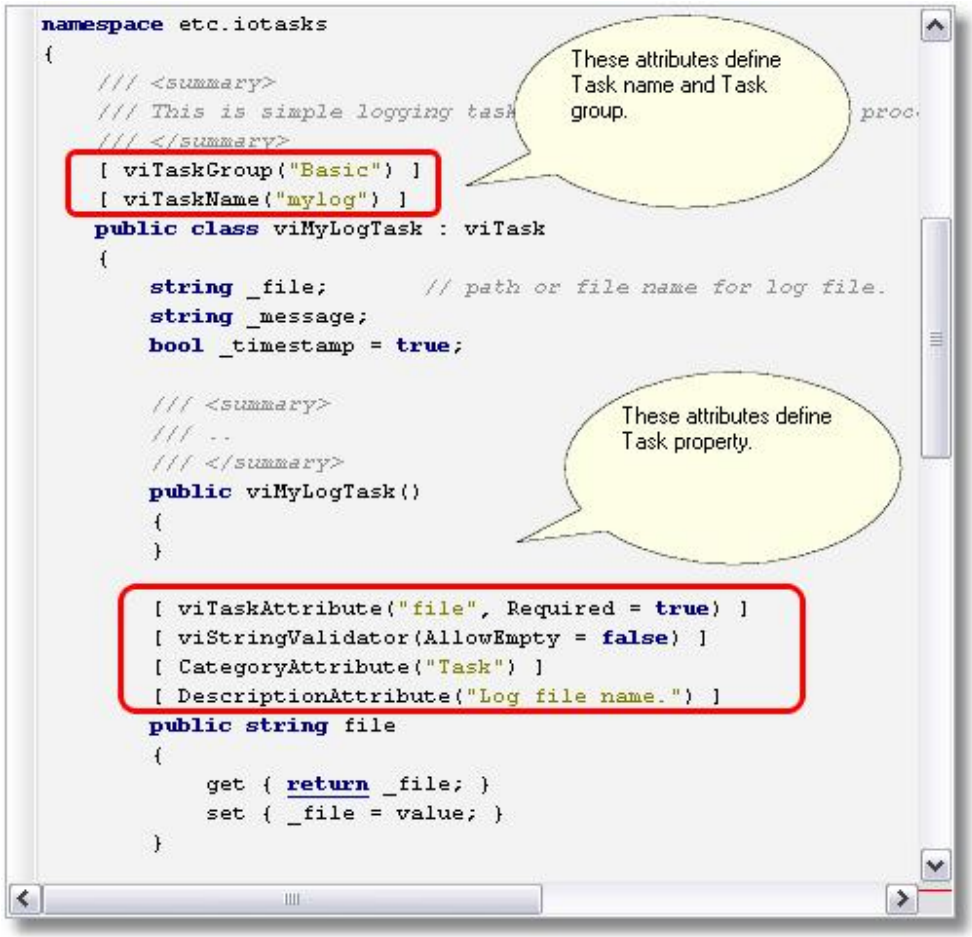
Developer SDK also provides example on how to write your own EchoHandler. EchoHandler is used to output information messages during job execution. If EchoHandler is not assigned to the job, information messages from <echo> and other tasks will not be sent to your application.

```
public class viFormEchoHandler : viEchoHandler
{
    private frmXmlScriptRun _form;

    public viFormEchoHandler(frmXmlScriptRun form)
    {
        _form = form;
    }

    public void Echo(string jobName, string message)
    {
        _form.SetText(string.Format("{0}, [{1}] {2}", DateTime.Now.ToString(),
            jobName, message));
    }
}
```

There is an example of EchoHandler that calls Form method every time there is new information message coming from the job.

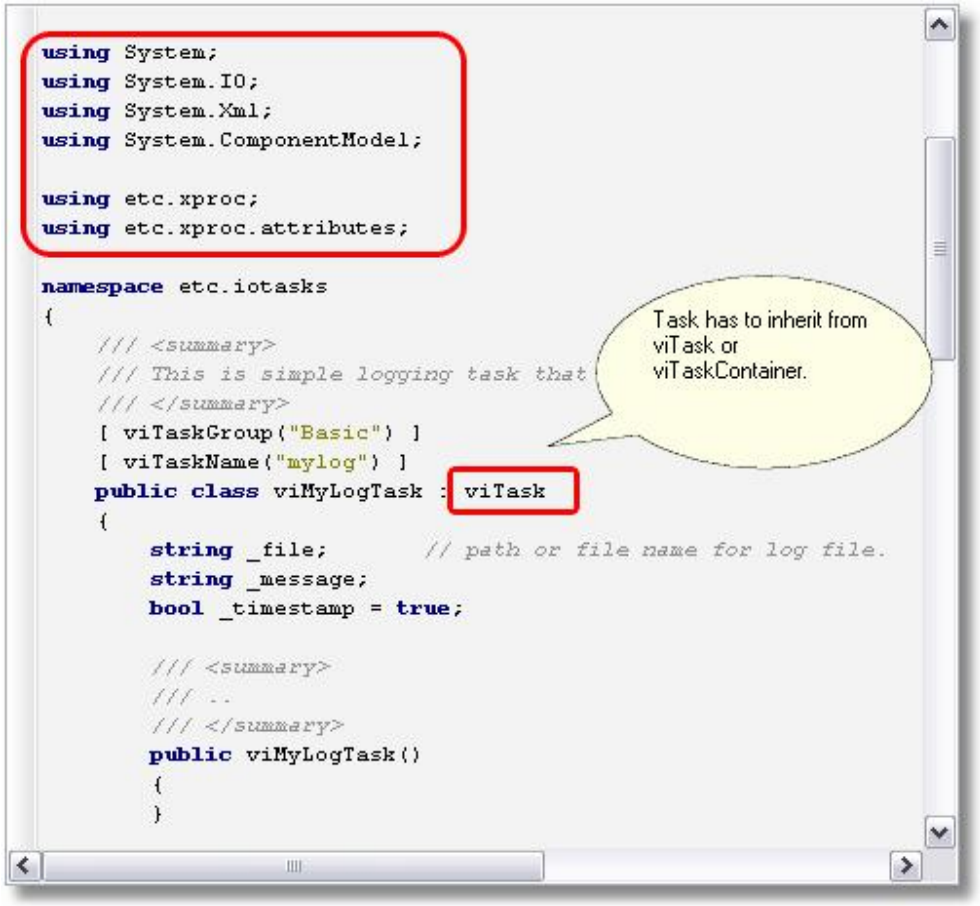
How to write your own Task

Tasks are classes inherited from viTask or viTaskContainer, decorated with special attributes and compiled into .NET library (CLR DLL). Tasks that can contain and execute other tasks nested below have to be inherited from viTaskContainer. Installation comes with sample task called viMyLogTask that you can modify, rename and repackage for your needs. It is simple task and can not contain other tasks nested in it. This sample task simply logs information messages passed to it via message property.




```
namespace etc.iotasks
{
    /// <summary>
    /// This is simple logging task                    proc
    /// </summary>
    [ viTaskGroup("Basic") ]
    [ viTaskName("mylog") ]
    public class viMyLogTask : viTask
    {
        string _file;       // path or file name for log file.
        string _message;
        bool _timestamp = true;

        /// <summary>
        /// ..
        /// </summary>
        public viMyLogTask()
        {
        }

        [ viTaskAttribute("file", Required = true) ]
        [ viStringValidator(AllowEmpty = false) ]
        [ CategoryAttribute("Task") ]
        [ DescriptionAttribute("Log file name.") ]
        public string file
        {
            get { return _file; }
            set { _file = value; }
        }
```

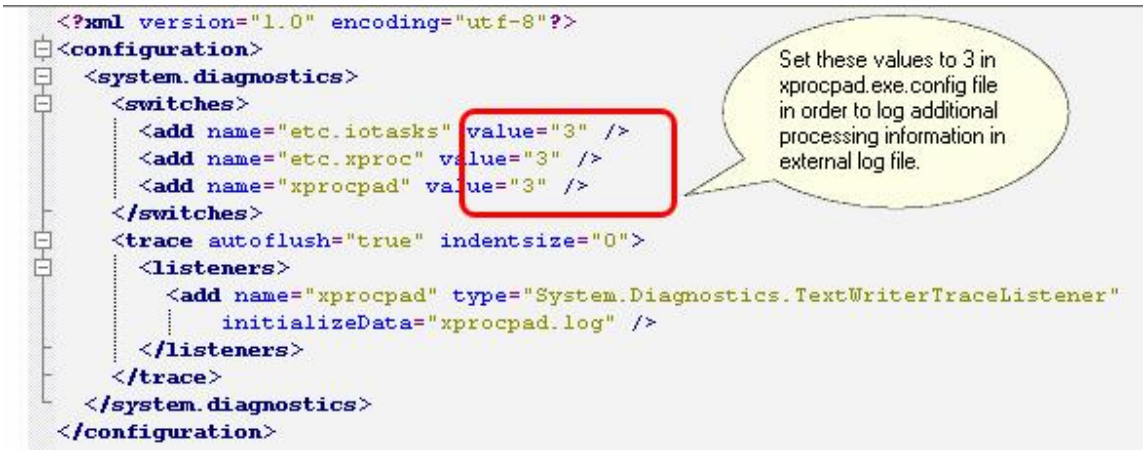
Your Task class and properties have to be decorated with special attributes in order to be properly displayed and accessible in editing tools. viTaskName attribute defines how task should be called and displayed in the graphical interface. Task name must be unique otherwise running script may call wrong task during execution.



Your Task class has to be inherited from viTask or viTaskContainer. In this example viMyLogTask is inherited from viTask and can not contain other nested tasks within itself. Developer SDK has other example called viMyLoopTask that inherits from viTaskContainer and can contain other tasks.

Technical Support

Please contact technical support if you have any questions or concerns. Contact information is listed on the product and company websites. For most technical support questions we will request processing log. This log can be obtained by changing log level values in xprocpad.exe.config file. Figure below shows values that need to be set before you run the processing script that fails in Script Editor. Processing will write extra information in xprocpad.log file that might be useful to identify technical problem.



These are special values to enable extensive logging into xprocpad.log file. Other values can reduce amount of logging produced. There are allowed values:

1 - errors only
2 - errors and warnings
3 - errors, warnings and additional information.