

[← Zurück zur Übersichtsseite](#)

Lösungshandbuch für die Migration von UNIX-Buildumgebungen – Kapitel 4: Migrieren des Buildsystems

(Engl. Originaltitel: [Chapter 4 - Migrating the Build System](#))

Einführung und Ziele

Bei einem Migrationsprojekt erstellt das Team in der Entwicklungsphase die Lösungskomponenten: Code und Infrastruktur sowie Dokumentation. Dabei wird der vorhandene Code so geändert, dass er in der neuen Umgebung funktionsfähig ist. Beim Erstellen des neuen Codes bleiben in der Regel einige Bestandteile des Originalsystems unverändert, wie zum Beispiel bereitgestellte APIs oder die Funktionsweise bestimmter Komponenten. Sowohl die Änderung des vorhandenen Codes als auch die Entwicklung neuen Codes werden in diesem Kontext als Migrationsaktivitäten angesehen. Aus diesem Grund wird die MSF-Entwicklungsphase in diesem Handbuch als Migrationsphase bezeichnet.

Obwohl die Entwicklungsarbeit in dieser Phase und diesem Leitfaden im Vordergrund steht, sind an der Erstellung und den Tests der Ergebnisse alle Teamrollen beteiligt. Von der Benutzerrolle werden beispielsweise die unter Umständen erforderlichen Schulungsunterlagen zusammengestellt. Einige Entwicklungsaufgaben können auch noch im Anschluss an die Tests in der Stabilisierungsphase fortgeführt werden.

Die Phase endet formal mit dem Meilenstein „Umfang vollständig“. Bei diesem grundlegenden Meilenstein erhält das Team vom Auftraggeber und den wichtigsten Projektbeteiligten die formale Bestätigung, dass alle Elemente der Lösung erstellt wurden sowie die Funktionen der Lösung gemäß den funktionalen Spezifikationen, die in der Planungsphase vereinbart wurden, vollständig sind.

Die wichtigsten Aufgaben dieser Phase sind in der folgenden Auflistung zusammengefasst:

- Entwickeln der Lösungskomponenten.
- Erstellen bzw. Migrieren der Systemtests, die in der Stabilisierungsphase verwendet werden sollen.
- Inkrementelles Erstellen der Lösung in mehreren täglichen Builds.
- Testen der Lösung (Durchführen von Tests für Codekomponenten, Datenbank, Sicherheit, Code- und Systemüberprüfung).

Entwickeln der Lösungskomponenten

Mit „Entwickeln der Lösungskomponenten“ ist bei der Migration des Buildsystems in erster Linie das Ermitteln und Portieren dieser Komponenten gemeint. Der Buildprozess weist in der Regel viele verschiedene Komponenten auf, wie zum Beispiel Makefiles, Anwendungen wie **sed** und **awk**, Shellskripts und einzelne Quelldateien, die beim Buildprozess kompiliert und ausgeführt werden. Dabei muss die wesentliche Funktionalität erhalten bleiben. Das bedeutet, dass die Module, Skripts und Tools die Aufgaben erledigen, die vom Buildsystem erfüllt werden müssen. In einigen Fällen werden neue Dateien oder Skripts entwickelt, die in der Windows-Umgebung spezielle Funktionen ausführen. Hinsichtlich der Systemstabilisierung wird die Zusammenarbeit aller Komponenten verbessert, damit die Ziele erreicht werden.

Bei der Migration von UNIX-Buildprozessen wird von einer Umgebung, die über eigene Tools verfügt, in eine andere Umgebung gewechselt. Die Tools kommen zwar weiterhin zum Einsatz, aber in einer unterschiedlichen Implementierung. Auch bei der Migration von UNIX (insbesondere Solaris) zu einem der drei UNIX-Umgebungsprodukte für Windows kann dies zu technischen Schwierigkeiten und Kompatibilitätsproblemen führen. Wie Sie diese Probleme lösen, erläutern die folgenden Abschnitte.

Dateisystemunterschiede

Die Windows- und UNIX-Dateisysteme weichen nicht nur hinsichtlich der Pfadtrennzeichen voneinander ab. Sie unterscheiden sich beispielsweise bei den jeweils unzulässigen Zeichen und bei der maximalen Länge von Datei- und Pfadnamen. Windows erlaubt die Angabe von Geräten (wie A:, C: usw.), während UNIX die Geräte in einem Dateisystem mit nur einem Stammverzeichnis versteckt.

Im Gegensatz zum FAT-Dateisystem weist das NTFS-Dateisystem unter Windows hinsichtlich Semantik und Verhalten viele Ähnlichkeiten mit UNIX auf, so dass die erforderlichen dateisystemspezifischen Änderungen bei einer Migration minimal sind. Das NTFS-Dateisystem unterstützt viele UNIX-Features, wie zum Beispiel Datei- und Verzeichnisnamen mit Unterscheidung von Groß-/Kleinschreibung, Berechtigungen für die Auflistung von Verzechnisinhalten, Aufteilung des Dateibesitzes in Benutzer und Gruppen, feste Verknüpfungen und ein Mechanismus für Dateizugriffsberechtigungen.

Groß- und Kleinschreibung

In UNIX-Dateisystemen wird zwischen Groß- und Kleinschreibung unterschieden. In Windows-Dateisystemen kann die Unterscheidung zwischen Groß-/Kleinschreibung beibehalten und unterschieden werden. Bei einem Dateisystem, in dem die Groß-/Kleinschreibung erhalten bleibt, kann eine Datei entweder **Makefile** oder **makefile** genannt werden, wobei die Schreibweise des Namens erhalten bleibt. Im Dateisystem dürfen aber nicht beide Namen vorhanden sein: Es darf keine gleichnamigen Dateien geben, die sich lediglich hinsichtlich Groß- und Kleinschreibung unterscheiden. Das Verhalten hängt vom Dateisystem und dem Subsystem für den Zugriff auf das Dateisystem ab. Wenn Dienstprogramme verwendet werden, die vom Interix-Subsystem abhängen, wird in FAT32-Dateisystemen und SMB-Netzwerkdateisystemen die Groß-/Kleinschreibung beibehalten, in lokalen NTFS-Dateisystemen wird sie gegebenenfalls unterschieden.

Wenn das Win32-Subsystem verwendet wird, bleibt die Groß-/Kleinschreibung in allen Dateisystemen erhalten. Bei der Suche von Dateien wird die Groß-/Kleinschreibung nicht als Kriterium verwendet. Mit anderen Worten: In einem Verzeichnis kann entweder **Makefile** oder **makefile** vorhanden sein, aber nicht beide Varianten gleichzeitig. Der Zugriff auf beide Dateien ist in beliebiger Schreibweise möglich, also beispielsweise über **MAKEFILE**, **Makefile** oder **makefile**.

Wenn Sie sich für ein Win32-basiertes UNIX-Toolset entscheiden, wie zum Beispiel das MKS Toolkit, oder wenn der Buildprozess in einem Netzwerkdateisystem funktionsfähig sein soll, dann muss der Buildprozess während des Migrationsprozesses überprüft werden. Des Weiteren sollte sichergestellt werden, dass alle Dateien in einem Verzeichnis unabhängig von der Groß-/Kleinschreibung (wie im Beispiel von **Makefile** und **makefile**) unterschiedliche Namen aufweisen.

Syntax von Pfadnamen

Die Syntax für Dateinamen unter Windows hängt von der ausgewählten UNIX-Umgebung ab.

In der Interix-UNIX-Umgebung wird nur die UNIX-Dateinamensyntax unterstützt. Die Interix-Umgebung stellt eine Dateisystemansicht mit einem einzigen Stammverzeichnis bereit (das Verzeichnis, in dem Interix installiert wurde). Der Zugriff auf andere Laufwerke mittels Laufwerkbuchstaben ist über das spezielle Verzeichnis **/dev/fs** möglich (beispielsweise **/dev/fs/C/Verzeichnis/Datei**). Auch Netzwerkfreigaben sind über das Verzeichnis **/net** (zum Beispiel **/net/Server/Freigabe/Verzeichnis/Datei**) verfügbar.

Cygwin vermittelt ebenfalls die Illusion eines Dateisystems mit einem einzigen Stammverzeichnis, dank einer eigenen Implementierung von Bereitstellungspunkten und einer eigenen Mounttabelle. Die

MKS-Umgebung unterstützt hingegen nicht das Konzept eines Dateisystems mit einem einzigen Stammverzeichnis. Bei der Migration in die MKS-Umgebung können daher absolute UNIX-Dateinamen zu Problemen führen. Um dieses Problem zu vermeiden, können symbolische Verknüpfungen für bekannte UNIX-Verzeichnisnamen erstellt werden, die auf das tatsächliche Verzeichnis innerhalb der MKS-Installation verweisen. Eine symbolische Verknüpfung mit **/bin** kann beispielsweise wie folgt eingerichtet werden:

```
ln -s $ROOTDIR/mksnt /bin
```

Die Umgebungen Cygwin und MKS unterstützen die Windows-Dateinamensyntax, einschließlich Laufwerkbuchstaben (**c:/Verzeichnis/Datei**) und UNC-Pfadnamen (**//Server/Freigabe/Verzeichnis/Datei**). Auch der normale Schrägstrich (/) sowie der umgekehrte Schrägstrich (\) als Trennzeichen werden unterstützt.

Reservierte Dateinamen

Im Win32-Subsystem sind bestimmte Dateinamen reserviert (die reservierten Namen gelten daher auch in den MKS- und Cygwin-Umgebungen). Die folgenden reservierten Bezeichnungen können nicht als Dateinamen verwendet werden: CON, PRN, AUX, CLOCK\$, NUL, COM1, COM2, COM3, COM4, COM5, COM6, COM7, COM8, COM9, LPT1, LPT2, LPT3, LPT4, LPT5, LPT6, LPT7, LPT8 und LPT9. Die Bezeichnungen sind weder als Dateinamensuffix noch als Dateinamextext zulässig. Namen wie **aux.c**, **Datei.aux** oder **NUL.txt** sind von daher nicht erlaubt.

Das Interix-Subsystem weist keine solchen reservierten Dateinamen oder Einschränkungen auf. Wenn Sie basierend auf den Interix-Tools eine Buildumgebung entwickeln und diese reservierten Dateinamen verwenden, müssen Sie beachten, dass diese Namen in der Win32-Umgebung oder Win32-Dienstprogrammen nicht zugänglich sind.

Einschränkungen der Zeichensätze für Dateinamen

Die zulässigen Zeichen für Datei- und Pfadnamen sind unter Windows eingeschränkter als unter UNIX. In Tabelle 4.1 sind die Zeichen aufgeführt, die im NTFS-Dateisystem in Windows-Dateinamen und -Pfadnamen nicht verwendet werden dürfen.

Tabelle 4.1: Unzulässige Zeichen in Windows-Datei- und -Pfadnamen

Zeichen	Name
^A .. ^_	ASCII-Steuerzeichen (Codewerte von 1 bis 31)
>	Größer als
<	Kleiner als
*	Sternchen
:	Doppelpunkt
"	Anführungszeichen
?	Fragezeichen
\	Umgekehrter Schrägstrich
	Senkrechter Strich
/	Schrägstrich

Das Interix-Subsystem besitzt weniger Einschränkungen. In Tabelle 4.2 sind die in Interix zusätzlich erlaubten Zeichen aufgeführt.

Tabelle 4.2: In Interix-Datei- und -Pfadnamen zusätzlich erlaubte Zeichen

Zeichen	Name
^A .. ^_	ASCII-Steuerzeichen (Codewerte von 1 bis 31)
*	Sternchen
:	Doppelpunkt
?	Fragezeichen
	Senkrechter Strich

Da Dateinamen im NTFS-Dateisystem UNICODE-codiert gespeichert werden, kann Interix diese speziellen Zeichen erkennen und speziellen reservierten UNICODE-Codes zuordnen. Diese Zuordnungen werden nur von der Interix-Umgebung unterstützt. In den Umgebungen Cygwin und MKS (Dienstprogramme, die auf dem Win32-Subsystem basieren) können diese Zeichen nicht angezeigt werden.

Der Doppelpunkt

Windows-Dateisysteme lassen Doppelpunkte nur in Dateinamen mit Verweis auf einen Laufwerksbuchstaben zu, der ein bereitgestelltes Dateisystem wie **c:/tmp** darstellt. Hinsichtlich des Doppelpunkts gibt es zwei Aspekte, die berücksichtigt werden müssen:

- Der Dateiname im ursprünglichen UNIX-Buildprozess enthält einen eingebetteten Doppelpunkt, wie zum Beispiel **:rofix**.
- Ein gültiger Windows-Dateiname mit einem Doppelpunkt (wie zum Beispiel **c:/tmp**) wird in einem Makefile als Zieldatei verwendet.

In der Interix-Umgebung ist der erste Aspekt unproblematisch, weil die Dateisystemsemantik des Interix-Subsystems nahezu mit der UNIX-Semantik übereinstimmt: Für Interix-basierte **make**-Dienstprogramme (**/bin/make**, **gmake**) können Doppelpunkte verwendet werden. Das Interix-Subsystem ordnet den Doppelpunkt automatisch einem speziellen Zeichen zu, den das Windows-Dateisystem akzeptiert. Der zweite Aspekt ist unzutreffend, weil Interix Windows-Pfadnamen (wie beispielsweise **c:/tmp**) nicht erkennt.

In der Cygwin-Umgebung birgt der erste Aspekt ein Problem, da die Windows-Umgebung in Dateinamen keine Doppelpunkte zulässt. Der zweite Aspekt ist unproblematisch, da das Ziel ein absoluter Pfadname unter Angabe eines Laufwerksbuchstabens sein kann.

In der MKS Toolkit-Umgebung stellt der erste Aspekt ein Problem dar, da Windows in Dateinamen keine Doppelpunkte zulässt. Der zweite Aspekt bringt ebenfalls ein Problem mit sich, und zwar durch die Methode, wie MKS-**make** bei der Suche nach Zieldateien das Makefile analysiert. Wenn der Zieldateiname einen Doppelpunkt enthält, wird dieser als Trennzeichen zwischen Zieldateien und vorausgesetzten Dateien behandelt. Um dieses Problem zu beheben, muss der Zielname in Anführungszeichen stehen. Beispiel:

```
"c:/tmp/a.exe" : "c:/tmp/main.obj"
```

Der Punkt

Das Win32-Subsystem kann keine Dateien verarbeiten, die mit einem Punkt (.) enden. In den Umgebungen MKS Toolkit oder Cygwin müssen solche Dateien umbenannt werden. Bei der Interix-Umgebung ist keine Änderung erforderlich.

Symbolische Verknüpfungen

Windows-Dateisysteme unterstützen keine symbolischen Verknüpfungen in der Weise, wie sie in UNIX-Systemen definiert werden. Windows stellt jedoch „Abzweigungspunkte“ bereit. Diese funktionieren nur in lokalen NTFS-Dateisystemen sowie nur mit Verzeichnissen und nicht mit Dateien. Mit Hilfe von Abzweigungspunkten kann ein Zielordner in einen anderen NTFS-Ordner „verpflanzt“ oder ein Datenträger in einem NTFS-Abzweigungspunkt bereitgestellt werden. Abzweigungspunkte sind für Programme transparent.

Der MKS Toolkit-Befehl **In** verwendet zum Implementieren von symbolischen Verknüpfungen NTFS-Abzweigungspunkte. Die Option **-s** kann nur für Verzeichnisse genutzt werden. Bei Verwendung der Option für eine Datei wird ein Fehler angezeigt.

Im Interix-Subsystem ist ein eigener spezieller Mechanismus implementiert, um symbolische UNIX-Verknüpfungen in beliebigen Dateisystemen (NTFS oder FAT, lokal oder im Netzwerk) zu ermöglichen. Bei den symbolisch verknüpften Dateien handelt es sich um spezielle Datendateien, die nur in der Interix-Umgebung als symbolische Verknüpfungen erkannt werden. Von Win32-Anwendungen werden sie nicht erkannt.

Der Cygwin-Befehl **In** unterstützt ebenfalls die Option **-s**, um symbolische Verknüpfungen zu erstellen. Dabei wird der „Verknüpfungsmechanismus“ von Windows verwendet. Bei der Quelldatei handelt es sich um eine spezielle Datendatei mit einem LNK-Suffix. Diese Dateien werden nur von solchen Win32-Anwendungen wie symbolische Verknüpfungen behandelt, die dieses spezielle LNK-Suffix erkennen, wie beispielsweise Explorer-Anwendungen. Die Verknüpfungen können in beliebigen Dateisystemen erstellt werden.

Feste Verknüpfungen

Bei einer festen Verknüpfung handelt es sich lediglich um eine andere Bezeichnung für einen Verzeichniseintrag, also der Zuordnung von Namen zu Dateien. Das NTFS-Dateisystem unterstützt das Vorhandensein mehrerer Verzeichniseinträge (oder mehrerer Namen) für die gleiche Datei. Das UNIX-Dienstprogramm, mit dem diese Verknüpfungen erstellt werden, ist **In**. Alle drei Produkte für die Portierung von UNIX-Umgebungen unterstützen diese Funktionalität.

Berechtigungen und Sicherheit

Die Modelle für Berechtigungen und Sicherheit in Windows und UNIX sind verschieden. Die Betriebssysteme Windows und UNIX verwenden unterschiedliche Mechanismen zur Benutzeridentifikation und Ressourcenzugriffssteuerung. Während in UNIX recht einfache IDs für Benutzer und Gruppen vorhanden sind, die durch einen numerischen 32-Bit-Wert dargestellt werden, handelt es sich bei den Sicherheits-IDs (SIDs) von Windows für Benutzer und Gruppen um numerische Werte variabler Länge, die normalerweise recht groß sind. Eine typische SID sieht als Zeichenfolge beispielsweise so aus: „S-1-5-21-1431262831-1455604309-1834353910-1000“. Für den Dateischutz verwendet UNIX neun Dateiberechtigungsbits mit jeweils drei Bits für Besitzer, Gruppe und andere Berechtigungen. Demgegenüber verwendet Windows DACLs (Discretionary Access Control Lists), die mehr als 14 verschiedene Zugriffsrechte unterstützen und jeweils einzelnen Benutzern oder Gruppen zugewiesen werden können.

Jede UNIX-Umgebung unter Windows realisiert das UNIX-Paradigma unter Windows anders. Die MKS-Dienstprogramme sind in dieser Hinsicht recht einfach, da sie genaue Benutzer- und Gruppen-IDs nur begrenzt unterstützen und geringe Zuordnungen zwischen Dateiberechtigungen und DACLs erlauben. Die Interix-Umgebung bietet die realistischste und genaueste Zuordnung mit eindeutigen IDs für lokale und globale Domänenkonten und einem genauen Übersetzungsmechanismus zwischen den UNIX-Dateiberechtigungsbits und DACLs. Die Cygwin-Umgebung implementiert die Zuordnung ähnlich wie Interix. Einige wenige Features sind in Cygwin allerdings weniger befriedigend gelöst.

In einem Buildprozess treten in der Regel kaum sicherheitsrelevante Probleme auf. An einigen wenigen Stellen können die Dienstprogramme **chmod**(1) oder **install**(1) vorkommen. Diese müssen überprüft werden, um festzustellen, ob die Sicherheitsanforderungen des Prozesses erfüllt sind. Wenn beispielsweise eine Anwendung in einem bestimmten Sicherheitskontext ausgeführt werden muss (zum Beispiel Anwendungen, die Benutzer-IDs festlegen können) und mit speziellen Berechtigungsbits installiert wird, müssen Sie prüfen, ob diese Art von Installation unter Windows weiterhin erforderlich ist. Ein weiteres Beispiel: Der Buildprozess selbst erstellt automatisch die Shellskripts, die ausgeführt werden müssen. Damit dieses Skript ausgeführt werden kann, wird vom Buildprozess unter Umständen das Dienstprogramm **chmod**(1) aufgerufen. In diesem Fall kann der einfache Befehl

```
chmod +x Skriptdatei
```

im Skript oder Makefile erforderlich sein. Alle UNIX-Umgebungen unter Windows unterstützen diesen Befehl.

In Buildsystemen kommt häufig das Dienstprogramm **install** vor. Damit werden Dateien oder Programme in ein gemeinsames Installationsverzeichnis verschoben oder kopiert. Darüber hinaus werden Dateiberechtigungen, Besitzer-ID und Gruppen-ID festgelegt (bei **install** treten häufig Sicherheits- und Berechtigungsprobleme auf). Alle UNIX-Umgebungen unter Windows unterstützen diesen Befehl. Er gehört zur Standarddistribution für Interix und Cygwin. Die MKS-Version befindet sich im Verzeichnis **samples/** auf dem Distributionsmedium.

Remotedateisysteme

Unter UNIX wird als Netzwerkdateisystem normalerweise NFS (Network File System) verwendet. Unter Windows ist CIFS (Common Internet File System) üblich, das auf SMB (Server Message Block) basiert.

Für eine Interoperation der Netzwerkdateisysteme gibt es NFS-Clients unter Windows und CIFS-Server unter UNIX. Selbst wenn das System so aufgebaut ist, dass die Quelldateien in einem UNIX-Dateisystem bleiben können und der Build auf einem Windows-System erfolgen kann, muss sichergestellt werden, dass die Dateinamen nach der Übersetzung durch einen NFS-Client oder CIFS mit Windows kompatibel sind. Die NFS- und CIFS-Software wandelt unzulässige Namen um. Wenn NFS-Clients auf den Windows-Computern bzw. CIFS-Server wie Samba auf dem UNIX-Rechner verwendet werden, muss der eigentliche Code nicht verschoben werden. Durch den Einsatz von NFS-Clients oder Samba verringern sich allerdings nicht die sonstigen Aufgaben in der Migration. Ein Remotedateisystem, das zwischen der Groß- und Kleinschreibung unterscheidet, kann beispielsweise eine wichtige Anforderung sein. Und zwar insbesondere dann, wenn Dateinamen wie **makefile** und **Makefile** im gleichen Verzeichnis vorkommen.

Ein weiteres Problem bei Remotedateisystemen besteht darin, dass die Systemzeiten stets synchronisiert sein müssen. Da das Dienstprogramm **make** in Bezug auf die Zeitstempel von Dateien sehr empfindlich ist, ist es unabdingbar, dass die Systemzeiten immer synchronisiert sind, da **make** sonst nicht die richtigen Voraussetzungen und Abhängigkeiten ermitteln kann. Glücklicherweise ist in allen Windows-Systemen seit Windows 2000 standardmäßig ein Zeitdienst enthalten. Systeme, die Bestandteil einer Domäne sind, werden automatisch mit dem Domänencontroller synchronisiert. Für andere Systeme, wie zum Beispiel UNIX-Systeme oder Windows-Systeme, die sich nicht in einer Domäne befinden, muss ein entsprechender Zeitserver eingerichtet werden.

Migrieren von Makefiles

Das Dienstprogramm **make** ist auf allen UNIX-Plattformen zu finden. Auf einer Plattform sind bisweilen mehrere verschiedene Versionen von **make** vorhanden. Unter Solaris gibt es beispielsweise drei Versionen: **/usr/ccs/bin/make**, **/usr/xpg4/bin/make** und **/usr/lib/svr4.make**. Letztere Version ist für Benutzer mit Makefiles vorgesehen, die von einem SystemVr4-basierten System portiert wurden. Die ersten beiden Versionen basieren auf der speziellen Implementierung von Sun Microsystems und unterstützen viele der Solaris-spezifischen Funktionen. Die zweite Version ist für Benutzer vorgesehen, die portierbare **make**-Umgebungen erstellen möchten, die auf den XPG- und POSIX-Spezifikationen basieren.

Auf den meisten UNIX-Plattformen ist ebenfalls die GNU-Implementierung von **make**, **gmake**, verfügbar. Diese Version von **make** weist die folgenden Merkmale auf:

- Einhaltung aller Spezifikationen des POSIX-Standards.
- Kompatibilität mit anderen Versionen von **make**, wie zum Beispiel die BSD 4.3- und System V-Versionen.
- Viele eigene proprietäre Features.

Wenn im Buildprozess bereits **gmake** verwendet wird, sollte das Programm nach Möglichkeit auch unter Windows eingesetzt werden (**gmake** steht in allen drei UNIX-Umgebungen zur Verfügung.) In den folgenden Abschnitten werden viele der Features in **make** behandelt, die in UNIX-Implementierungen und Windows-Versionen unterschiedlich sind. So können Sie rechtzeitig auf mögliche Problembereiche in der Migration von Buildumgebungen aufmerksam werden.

Start von „make“

Beim Ausführen von **make** wird zuerst eine Datei mit allen Standardregeln und vordefinierten Makrodefinitionen gelesen. Diese Datei wird auch als Start- oder Standardmakefile bezeichnet. Um die Makefiles zu migrieren, müssen unter Windows die Startmakefiles um spezielle Regeln ergänzt werden.

Bei der MKS-Version von **make** lautet diese Startdatei standardmäßig **\$ROOTDIR/etc/startup.mk**. Diese Datei kann von der Umgebungsvariablen MAKESTARTUP außer Kraft gesetzt werden. Bei der Installation des MKS Toolkit for Developers oder des MKS Toolkit for Enterprise Developers wird diese Umgebungsvariable beispielsweise auf **\$ROOTDIR/etc/nutc.mk** festgelegt.

Bei der Interix-Version von **make** lautet diese Datei standardmäßig **/usr/share/mk/sys.mk**. Interix-**make** verwendet ebenfalls die Umgebungsvariable MAKESTARTUP, damit Benutzer die Standarddatei außer Kraft setzen können.

Auf Solaris-Systemen sucht **make** zuerst nach **./make.rules** und dann nach den Standarddateien in **/usr/share/lib/make/make.rules**. Gelesen wird immer die zuerst gefundene Datei.

Bei **gmake** sind die Standardregeln im Programm integriert und können nicht wie bei den anderen Versionen von **make** eingelesen werden.

Einbinden anderer Makefiles

In der Solaris-Version von **make** kann mit der Anweisung **include** eine andere Datei angegeben werden, die so verarbeitet werden soll, als ob ihr Inhalt in der entsprechenden Zeile des Makefiles enthalten wäre. Die ersten sieben Buchstaben einer Zeile müssen das Wort **include** bilden, und dahinter muss ein Leer- oder Tabulatorzeichen folgen.

Diese Syntax wird auch von den MKS- und Interix-Versionen von **make** sowie **gmake** unterstützt.

Mehrzeilige Kommentare

Ein Kommentar beginnt seit jeher mit einem Gattersymbol (#) und endet bei einem nicht geschützten Zeilenumbruch. Beispiel:

```
# Dies ist ein mehrzeiliger Kommentar. \  
    Zeile 2 des Kommentars \  
    Zeile 3 endet mit einem nicht geschützten Zeilenumbruch und beendet den Kommentar.
```

Diese Definition wird auch durch die POSIX- und UNIX-Standards vorgeschrieben. Die Syntax wird von den Solaris- und MKS-Versionen von **make** sowie **gmake** unterstützt, nicht aber von Interix-**make**. Eine andere Möglichkeit, mehrzeilige Kommentare zu erstellen, besteht darin, am Anfang jeder Zeile ein #-Zeichen einzufügen.

Makros

Bei Makros handelt es sich um einfache Variablen, wobei dem Makronamen eine Zeichenkette (String) zugeordnet wird. Makros können im Makefile an beliebiger Stelle vorkommen. Die Syntax einer Makrodefinition (oder -zuweisung) lautet:

```
VAR = Zeichenkette
```

Ein Makrowert kann sich über mehrere Zeilen erstrecken, wenn am Ende ein umgekehrter Schrägstrich steht (geschützter Zeilenumbruch). In der Regel (aber nicht zwingend) beginnt die Fortsetzungszeile mit einem Leerzeichen, da dadurch die Lesbarkeit verbessert wird.

Der Verweis auf ein Makro erfolgt normalerweise mit der Syntax $\$(\text{Makroname})$. Auf einen Makronamen mit einem einzigen Buchstaben kann ohne Klammern verwiesen werden: $\$x$. Wenn auf Makros verwiesen wird, werden sie in ihre Werte erweitert. Diese Erweiterung hängt von der Position des Makros im Makefile ab.

Die Standardmakrosyntax ist in allen Implementierungen von **make** einheitlich. Die einzelnen Implementierungen unterstützen unter Umständen jedoch andere Features. Einige dieser Funktionen werden in den folgenden Abschnitten beschrieben.

Spezielle Makronamen

Bestimmte Makronamen werden von **make** speziell interpretiert. Die verschiedenen Versionen von **make** weisen eigene Spezialmakros auf, die nun folgenden sind allen gemeinsam:

- SHELL
Bei dem SHELL-Makro handelt es sich um ein Spezialmakro, dessen Wert nicht von der Umgebung geerbt wird. Es kann nur im Makefile festgelegt werden. Der Wert dieses Makros gibt das Programm an, das zum Ausführen von Anweisungsbefehlen verwendet werden soll, und wird normalerweise auf einen vollständigen Pfadnamen festgelegt (wie zum Beispiel **/bin/sh**). Bei der Verwendung dieses Features im UNIX-Makefile muss der vollständige Pfadname in der UNIX-Umgebung unter Windows vorhanden sein. Ansonsten muss dieser Wert geändert oder das Makro entfernt werden.
Bei der MKS-Version von **make** funktioniert der Wert **/bin/sh** nur dann, wenn im System ein Pfad mit dem Namen **/bin/sh** angelegt wurde. Sie können dieses Verzeichnis erstellen und alle erforderlichen Dienstprogramme (wie **sh**) aus **\$ROOTDIR/mksnt** kopieren. Bei einem lokalen NTFS-Dateisystem können Sie auch eine symbolische Verknüpfung zu **\$ROOTDIR/mksnt** erstellen, zum Beispiel **ln -s \$ROOTDIR/mksnt /bin**. Dieser Pfadname muss in jedem Dateisystem erstellt werden, von dem aus **make** aufgerufen wird, da das Stammverzeichnis (/) relativ zum Dateisystem des aktuellen Arbeitsverzeichnisses ist.
Eine alternative Lösung besteht darin, den Wert des SHELL-Makros in **\$MKSBIN/sh.exe** oder **\$ROOTDIR/mksnt/sh.exe** zu ändern.
- MAKEFLAGS
MAKEFLAGS enthält eine Liste von **make**-Befehlsflags sowie Makrozuweisungen. Diese werden in **make** so verwendet, als befänden sie sich in der Befehlszeile, und werden bei jedem rekursiven Aufruf von **make** übergeben. Einige Befehlsflags sind in MAKEFLAGS nicht zulässig, da sie in diesem Kontext sinnlos sind. Entsprechende Informationen finden Sie auf der Manpage zu **make(1)**.
Dieses Spezialmakro wird von **make** erstellt und enthält die Befehlsflags und Makrodefinitionen, die vom **make**-Befehl verwendet werden. Wenn die MAKEFLAGS-Umgebungsvariable gesetzt ist, interpretiert **make** die Werte wie Befehlszeilenoptionen, und zwar zusätzlich zu den in der Befehlszeile angegebenen Optionen. Beim Start von **make** wird das MAKEFLAGS-Makro erstellt, das alle Optionen und Makrodefinitionen enthält, die in der Befehlszeile angegeben wurden sowie in der Umgebungsvariablen MAKEFLAGS enthalten sind. Dieser MAKEFLAGS-Makrowert wird immer in die Umgebung exportiert, so dass verschachtelte **make**-Befehle die Optionen und Makrodefinitionen übernehmen, mit denen der übergeordnete **make**-Befehl aufgerufen wurde.
Vor MAKEFLAGS wurde in den früheren UNIX-Versionen von **make** eine Variable mit dem Namen MFLAGS verwendet, in der lediglich die Optionsargumente und keine Makrodefinitionen enthalten waren. Viele aktuelle Versionen von **make** unterstützen aus Gründen der Abwärtskompatibilität weiterhin MFLAGS.
Das Format der in MAKEFLAGS gespeicherten Werte ist in den verschiedenen Versionen von **make** nicht einheitlich. Einige Versionen verketteten alle Optionen mit oder ohne führenden Bindestrich (zum Beispiel **-ek** oder **ek**), während andere Versionen jede Option mit einem führenden Bindestrich verwalten (zum Beispiel **-e -k**). Beachten Sie weiterhin, dass die MKS-Version von **make** in MAKEFLAGS keine Makrodefinitionen speichert, die in der Befehlszeile definiert wurden, sondern lediglich die Optionen. Wenn **make** verschachtelt (oder rekursiv) aufgerufen wird, müssen die Makefiles in der MKS-Umgebung so geändert werden, dass explizit alle Makrodefinitionen enthalten sind, die an die nächste **make**-Ebene übergeben werden müssen. Dies wird normalerweise von MAKEFLAGS automatisch geändert.

Makrozuweisungen

Einem Makro wird normalerweise wie folgt ein Wert zugewiesen:

```
VAR = Wert
```

Dabei wird der Wert von *Wert* ausgewertet und erweitert, wenn das Makro zum ersten Mal *verwendet* wird, und nicht bei der Zuweisung des Wertes.

Die Makrozuweisung kann auch mit der folgenden Syntax erfolgen:

```
VAR := Wert
```

Dieser Zuweisungsoperator wird von den meisten Versionen von **make** unterstützt. Er hat aber in der Solaris-Version eine andere Bedeutung als in den übrigen. In den meisten Versionen führt dieser Operator dazu, dass der Wert bei der Definition und nicht bei der Verwendung analysiert und erweitert wird. In Solaris wird diese Syntax als bedingte Makrozuweisung interpretiert. Diese Art von Zuordnung wird weiter unten in diesem Kapitel im Abschnitt *Bedingte Makrodefinitionen* beschrieben.

Ersetzen von Befehlen in Makrozuweisungen

In einigen Implementierungen von **make** gibt es spezielle Zuordnungstypen. Der **make**-Befehl in Solaris unterstützt beispielsweise Makrozuweisungen für das Ersetzen von Befehlen. Bei der Wertezuweisung für das Makro wird eine Shellbefehlszeile ausgeführt und das Ergebnis der Variablen zugewiesen. Die entsprechende Syntax lautet:

```
VAR:sh = Shellbefehlszeile
```

Diese Syntax wird auch von der Interix-Version von **make**, nicht aber von MKS-**make** und **gmake** unterstützt. In MKS-**make** können Sie diese Zuweisung ersetzen, indem Sie das Ergebnis von *Shellbefehlszeile* vor dem Aufrufen von **make** erfassen und den Wert in einer Befehlszeilenmakrodefinition übergeben, zum Beispiel:

```
make VAR="$(Shellbefehlszeile)" <weitere Operatoren . . .>
```

Bei **gmake** kann die spezielle **shell**-Funktion verwendet werden, die in **gmake** zur Verfügung steht. In diesem Fall kann die Makrozuweisung im Makefile folgendermaßen umgeschrieben werden:

```
VAR := $(shell Shellbefehlszeile)
```

Ersetzen von Befehlen in Makroverweisen

Wenn auf Makros verwiesen wird, werden deren Werte anstelle des Makros eingesetzt. Das folgende Beispiel veranschaulicht, wie ein Makro durch den Wert eines Shellbefehls ersetzt werden kann. Bei dem Verweis auf das Makro **CMD** wird der Shellbefehl **cat Objektdateiliste** ausgeführt und das Ergebnis dem Wert von **CMD** zugewiesen.

```
CMD = cat Objektdateiliste
```

```
program : ${CMD:sh}
```

```
cc -o $@ $?
```

Wenn die Datei **Objektdateiliste** die Dateien **hello.o** **goodbye.o** enthält, hängt das Zielprogramm hier von diesem Objektdateinamen ab und wird mit Hilfe dieser Objektdateien erstellt.

Dieses Feature wird von den Solaris- und Interix-Versionen von **make** unterstützt, jedoch nicht von **gmake** und der MKS-Version von **make**.

Makromodifizierer

Makros werden normalerweise erweitert, wenn auf das Makro verwiesen wird. Nachdem Makros erweitert wurden, können die Werte in der Erweiterung auf verschiedene Weise verändert werden. Diese Modifizierersyntax hängt von der jeweiligen Version von **make** ab und stellt bei der Migration unter Umständen ein Problem dar.

Die Syntax für Makromodifizierer lautet:

```
VAR:Modifizierer[:Modifizierer ...]
```

Es gibt zwei Modifiziererformate, die umfassend unterstützt werden. Die erste Syntax lautet:

```
Altes_Suffix=Neues_Suffix
```

Alle Versionen von **make** unterstützen diese Modifizierersyntax. Hierbei handelt es sich um einen Suffixersatzmodifizierer. In jedem Wort, das das Suffix **Altes_Suffix** enthält, wird **Altes_Suffix** durch **Neues_Suffix** ersetzt. Beispiel:

```
OBJS=${SRCS:.c=.o}
```

konvertiert eine Liste von C-Quelldateinamen in der Variablen **SRCS** in eine Liste von Objektdateinamen, die anschließend der Variablen **OBJS** zugewiesen wird.

Das zweite Modifiziererformat lautet:

```
Präfix%Suffix = Zeichenfolge1%Zeichenfolge2
```

Hierbei handelt es sich um einen Musterersatzmodifizierer. Das Prozentzeichen (%) auf der linken Seite des Gleichheitsoperators (=) steht für eine beliebige Zeichenfolge. Wenn ein Wort sowohl Präfix als auch Suffix enthält, steht % für alle dazwischenliegenden Zeichen. Diese Zeichenfolge wird in allen Vorkommen von % auf der rechten Seite eingesetzt. Beispiel:

```
FULLNAME=/usr/local/bin/perl
```

```
SUBDIRS=${FULLNAME:/usr%/perl=%}
```

Dadurch wird im Wert von **SUBDIRS** die Zeichenfolge **local/bin** eingesetzt.

Auf der rechten Seite kann das Zeichen % beliebig oft vorkommen.

Bedingte Makrodefinitionen

Bei einer bedingten Makrodefinition wird einer Variablen nur ein Wert zugewiesen, wenn **make** Zielfile aus einer bestimmten Liste verarbeitet. Diese bedingte Definition gibt es nur in der Solaris-Version von **make**. Sie weist die folgende Form auf:

```
Zielliste :=VAR = Wert
```

Diese Definition weist den Wert der Variablen *VAR* zu, wenn **make** die Zielfile mit dem Namen **Zielliste** und deren abhängige Dateien verarbeitet. Die Makrodefinition wird nur bei der Verarbeitung der betreffenden Zielfile und deren abhängigen Dateien verarbeitet.

Suchpfade

In vielen umfangreichen Buildsystemen werden die Quelldateien in einem oder mehreren verschiedenen Verzeichnissen gespeichert. Das Dienstprogramm **make** bietet einen Mechanismus, mit dem die Zieldateien, abhängigen Dateien oder `.INCLUDE`-Dateien in einem oder mehreren Verzeichnissen gesucht werden können. Die meisten Implementierungen unterstützen das `VPATH`-Makro. Der Wert dieses speziellen Makros gibt eine Liste von Verzeichnissen an, die **make** durchsuchen soll. Das Format dieser Verzeichnisliste stimmt mit dem der Umgebungsvariablen `PATH` überein: Die einzelnen Verzeichnisse sind durch Doppelpunkte (`:`) voneinander getrennt.

Die einzige bekannte Implementierung, die dieses Makro nicht unterstützt, ist die MKS-Version von **make**, die eine geringfügig abweichende Syntax benutzt. Anstelle eines Makros wird eine spezielle Zieldatei mit dem Namen `.SOURCE` verwendet. Die `.SOURCE` zugeordnete Liste abhängiger Dateien ist die Liste der Verzeichnisse, die **make** durchsuchen soll.

Wenn Sie bei einer Portierung von UNIX zum MKS Toolkit das `VPATH`-Makro verwenden, muss das Makefile so geändert werden, dass es `.SOURCE` benutzt. Ändern Sie beispielsweise das Makefile von:

```
VPATH = src:headers:../othersrc
```

in:

```
.SOURCE: src headers ../othersrc
```

Standardregeln und Standardmakrowerte

Beim Start von **make** wird zunächst u. a. eine Liste der Standardregeln (zum Beispiel Suffixregeln) erstellt, und es werden einige Standardmakronamen initialisiert.

Bei Solaris, MKS und Interix befinden sich diese Definitionen in den Standardregel- oder Standardstartdateien (siehe Abschnitt *Start von "Make"* weiter oben in diesem Kapitel). Bei **gmake** sind diese Definitionen fest in der Binärdatei codiert.

Viele dieser konventionellen Makros und Suffixregeln sind in den meisten Implementierungen von **make** identisch. In der Solaris-Version von **make** und in **gmake** sind mehr Makros und Regeln definiert. Namen wie **COMPILE.c** und **LINK.c** sind in Solaris und **gmake** verfügbar, aber nicht in Interix oder MKS. Bei einer Migration von Solaris muss überprüft werden, ob alle vordefinierten, benötigten Makros in der ausgewählten Zielimplementierung vorhanden sind. Andernfalls müssen Sie sie zu Ihrer eigenen Umgebung hinzufügen.

Dateinamensuffixe

Unter UNIX hat das Format von Dateinamen keine besondere Bedeutung. Alle speziellen Formate sind lediglich historische Konventionen. Diese Konventionen sind in der Regel nicht bindend vorgeschrieben und für UNIX-Anwendungen oder das UNIX-System belanglos. Objektdateien enden normalerweise mit der Dateierweiterung `.o`, sie könnte aber ebenso gut `.obj` oder `.object` lauten. Die Dateinamensyntax ist für den C-Compiler oder Linker belanglos. Es wird lediglich überprüft, ob die Datei ein gültiges Objektdateiformat enthält.

Unter Windows haben Dateinamensuffixe eine besondere Bedeutung. Viele Windows-Anwendungen ordnen Dateien mit speziellen Suffixen bestimmte Verhaltensweisen zu. Bei der Anwendungs- oder Konstruktionsprozessmigration werden Sie unter Umständen feststellen, dass Dateinamen mit Windows-spezifischen Suffixen erstellt werden müssen, damit sie von anderen Windows-Anwendungen ordnungsgemäß erkannt werden. Dies kann Auswirkungen auf die Makefiles oder Konfigurationsskripts haben.

Die Shells im MKS Toolkit führen eine ausführbare Binärdatei nur dann aus, wenn die Datei die Dateierweiterung **.exe** aufweist. Andernfalls wird die Datei als Shellskript interpretiert, und es kommt zu einem Fehler. Im eigentlichen Buildprozess stellt das normalerweise kein Problem dar, da die ausführbaren Dateien in der Regel durch die Befehle **cc** oder **gcc** erstellt werden und diese Compiler automatisch ausführbare Dateien mit dem Suffix **.exe** anlegen. Im MKS Toolkit und in Cygwin erstellt der Befehl **cc -o hello hello.c** (**cc** bei Cygwin durch **gcc** ersetzen) beispielsweise eine Datei mit dem Namen **hello.exe**. Obwohl explizit eine ausführbare Datei ohne Suffix angegeben wurde, wird eine Datei mit dem richtigen Suffix angelegt.

Weitaus problematischer sind Shellskripts oder Makefiles, in denen explizit auf die Namen von ausführbaren Dateien verwiesen wird. Der Dateiname, auf den verwiesen wird, entspricht wegen des neuen **.exe**-Suffixes unter Umständen nicht dem Namen der erzeugten ausführbaren Datei. In Makefiles sind davon meistens Zieldateinamen betroffen. Beispiel: Die Regel

```
hello:
```

```
cc -o hello hello.c
```

führt dazu, dass die Datei **hello.exe** erstellt wird. **hello.exe** ist aber nicht der Zieldateiname, sondern **hello**. Da die Zieldatei **hello** niemals erstellt wird, führt **make** ständig diese Anweisung aus.

In einigen Fällen (wie bei Cygwin) wurde das Dienstprogramm **gmake** so geändert, dass dieser Sonderfall automatisch verarbeitet wird und es zu keiner fehlenden Übereinstimmung zwischen **hello** und **hello.exe** kommt.

Bei Interix gibt es ein anderes Problem mit Dateisuffixen. Das Compilerskript **wcc** erstellt keine ausführbaren Dateien mit der Dateierweiterung **.exe**. Dies ist auch sinnvoll, denn alle Build- und Konstruktionstools verhalten sich so weiterhin richtig, weil alle vorausgesetzten und abhängigen Dateien hinsichtlich der Dateinamensyntax unverändert bleiben. Das kann aber bedeuten, dass die Binärdatei der endgültigen Windows-Anwendung nicht das richtige Windows-Dateinamenformat aufweist und daher von anderen Windows-Anwendungen (wie Explorer) nicht ausgeführt werden kann. In diesem Fall kann das Makefile um einen neuen Befehl ergänzt werden, der die Datei umbenennt. Sie können auch ein Skript programmieren, das nach dem Build ausgeführt wird und die endgültigen Binärdateien in ein geeignetes Dateinamenformat bringt.

Implizite Regeln

Implizite Regeln bilden die Eckpfeiler von **make**: Sie ordnen einer Zieldatei eine vorausgesetzte Datei zu, indem bestimmte Dateinamen abhängig von allgemeinen Mustern im Dateinamen zugewiesen werden. Es gibt zwei gebräuchliche Arten von impliziten Regeln: Suffixregeln und Musterübereinstimmungsregeln. Alle Implementierungen von **make** unterstützen Suffixregeln. Diese basieren auf der Zuordnung von Mustern für Dateinamenerweiterungen.

Einige Implementierungen unterstützen flexiblere Musterübereinstimmungsregeln, die eine Musterzuordnung an beliebiger Stelle im Dateinamen und nicht nur im Suffix zulassen. Bisweilen wird diese Art von Regel Metaregel genannt.

Suffixregeln

Jede **make**-Implementierung verfügt über mehrere implizite Suffixregeln. Diese Regeln definieren, wie die verschiedenen Dateitypen erstellt und gegebenenfalls aus einem anderen Typ transformiert werden. Die Dateien werden anhand ihres Dateinamensuffixes identifiziert. Das Dienstprogramm **make** ist eines der wenigen UNIX-Dienstprogramme, das Dateinamensuffixe einsetzt.

Der Mechanismus für Suffixregeln basiert auf Dateinamenerweiterungen, die in der UNIX-Community standardisiert sind und anhand derer die verschiedenen Dateitypen bei der Quellcodeentwicklung und Kompilierung identifiziert werden.

Die Definitionen aller Standardsuffixregeln befinden sich normalerweise in der **make**-Startkonfigurationsdatei. Sie müssen gewährleisten, dass das unter Windows verwendete **make**-Programm über die richtigen impliziten Regeln verfügt, damit Dateien ohne Suffixe erstellt werden. Beispiel: Die folgende Suffixregel

```
.c:
$(CC) -o $@ $(CFLAGS) $(LDFLAGS) $<
```

kann verwendet werden, um die Zieldatei `hello` mit der folgenden Befehlszeile anzulegen:

```
hello: hello.c
```

In der MKS-Version von **make** fehlt diese implizite Regel allerdings. Ohne diese Regel weiß MKS-**make** nicht, wie die Datei `hello` aus `hello.c` erstellt werden soll. Diese Regel kann leicht zum Makefile hinzugefügt werden. Dadurch werden aber nicht alle Probleme gelöst. Beim MKS Toolkit besteht weiterhin das Problem, dass `cc` eine Datei mit dem Namen `hello.exe` (nicht `hello`) erstellt und **make** nicht erkennt, dass diese beiden Dateien äquivalent sind. Bei jeder Ausführung von **make** wird der Zieldateiname `hello` nicht gefunden, so dass die implizite Regel immer ausgeführt wird.

Um die verschiedenen Suffixe zwischen UNIX und Windows zu verarbeiten, definiert die **make**-Startdatei von MKS einige einfache Makros, wie `$E`, `$O`, `$S` und `$A`, die den konventionellen Windows-Suffixen `.exe`, `.obj`, `.s` bzw. `.lib` zugeordnet sind:

```
E = .exe
S = .s
O = .obj
A = .lib
```

Wenn die Makefiles in Windows und UNIX gemeinsam genutzt werden sollen, können Sie die Vorkommen dieser Suffixe im Makefile durch diese Makros ersetzen und in der **make**-Startdatei für UNIX bedingte Definitionen für diese Makros mit den entsprechenden Werten für UNIX festlegen. Beispiel:

```
E =
S = .s
O = .o
A = .a
```

Dieses Verfahren ist nicht auf die Umgebung von MKS-**make** beschränkt, sondern kann in allen anderen **make**-Umgebungen verwendet werden.

Musterübereinstimmungsregeln

Mit Musterübereinstimmungsregeln kann basierend auf Dateinamenpräfixen und/oder -suffixen eine Beziehung zwischen einer Zielfile und einer abhängigen Datei angegeben werden. Ein Beispiel für diese Art von Regel ist:

```
% : RCS/%,v  
  
co -1 $<
```

Musterübereinstimmungsregeln wurden vor vielen Jahren in die Solaris-Version von **make** eingeführt. Seitdem wurden solche Regeln auch in die MKS- und GNU-Versionen von **make** integriert. Interix-**make** unterstützt diesen Mechanismus nicht.

Bibliotheks- und Archivunterstützung

Einige Versionen von UNIX-**make**, wie zum Beispiel Solaris- und System V-basierte Implementierungen, unterstützen eine Syntax, um abhängige Dateien zu ermitteln, die in Archiven gespeichert sind. Archive werden mit dem Dienstprogramm **ar(1)** erstellt und enthalten mehrere Dateien, die so genannten Members. Meistens werden Archive zum Speichern von Objektdateien genutzt, die bei der Kompilierung verwendet werden.

Historische Versionen von **make** unterstützen eine Syntax, mit der Archivmembers als Zielfile oder vorausgesetzte Datei angegeben werden können. Die Members eines Archivs werden mit der folgenden Syntax angegeben:

```
archive(Member [Member ...]),, zum Beispiel example.a(member1.o member2.o).
```

Wenn ein Makefile vorhanden ist, das beispielsweise

```
example.a : example.a(member1.o member2.o)
```

enthält, und Sie den folgenden Befehl eingeben:

```
make example.a
```

erstellt **make** die Dateien **member1.o** und **member2.o** (mit einer impliziten Regel wie **.c.o**, vorausgesetzt die Quelldateien sind verfügbar) und anschließend das Archiv **example.a**, in das diese beiden **.o**-Dateien eingefügt werden.

Diese Syntax wird von allen **gmake**-Versionen und Interix-**make** unterstützt, nicht jedoch von MKS-**make**.

Bei der Interix-Version von **make** besteht ein Problem hinsichtlich der impliziten Regeln **.o.a** und **.c.a**, die sich in der Datei **/usr/share/mk/sys.mk** befinden. Diese impliziten Regeln werden bei der Auflösung der Beziehungen der abhängigen Archivdateien verwendet.

Die Regeln sehen normalerweise wie folgt aus:

```
.o.a:
$(AR) $(ARFLAGS) $@ $*.o

rm -f $*.o
```

```
.c.a:
$(CC) -c $(CFLAGS) $<

$(AR) $(ARFLAGS) $@ $*.o

rm -f $*.o
```

Sie können aber folgendermaßen korrigiert werden:

```
.o.a:
$(AR) $(ARFLAGS) ${.ARCHIVE} $*.o

rm -f $*.o
```

```
.c.a:
$(CC) -c $(CFLAGS) $<

$(AR) $(ARFLAGS) ${.ARCHIVE} $*.o

rm -f $*.o
```

Versionskontrolle (RCS, SCCS)

Versionskontrolle und Konfigurationsverwaltung erfolgen in der Regel getrennt und unabhängig von der Buildverwaltung. Es ist wesentlich einfacher, separate Tools zu erstellen, als für **make** einwandfrei funktionierende, implizite Versionskontrollregeln aufzustellen. Wenn die Quelldateien von **make** automatisch aktualisiert werden dürfen, gibt es einige grundlegende Probleme, insbesondere wenn mehrere Personen die gleichen Dateien bearbeiten. Von daher sollte aus Sicherheitsgründen die Verantwortung, vor dem Build die richtigen Dateiversionen zu ermitteln, dem Programmierer übertragen werden.

In die historischen Versionen von UNIX-**make** war eine spezielle Unterstützung für SCCS (Source Code Control System) integriert, ein recht altes Versionskontrollsystem, das in System V entwickelt wurde. Obwohl SCCS noch heute eingesetzt wird, werden zunehmend modernere Tools für Versionskontrolle und Konfigurationsverwaltung verwendet, die SCCS ablösen.

Die Unterstützung für SCCS musste in **make** auf andere Weise als die übrigen Regeln integriert werden, da SCCS die Dateien über Präfixe und nicht über Suffixe verwaltet. Wenn beispielsweise eine C-Quelldatei mit dem Namen **hello.c** der Kontrolle von SCCS unterliegt, wird eine Datei mit dem Präfix **s.** (**s.hello.c**) erstellt. **gmake** unterstützt SCCS in gewissem Umfang, bei den Interix- und MKS-Versionen von **make** ist das nicht der Fall.

Die Migration von einem UNIX-System, in dem SCCS verwendet wurde, nach Windows stellt Sie vor ein Problem, da die SCCS-Tools in den MKS Toolkit-, Interix- oder Cygwin-Produkten nicht verfügbar sind. SCCS ist zwar nicht mehr weit verbreitet, eventuell aber auf Ihrem UNIX-System vorhanden. Wenn SCCS erforderlich ist, gibt es einige mögliche Lösungen. Es existieren mehrere alternative SCCS-Implementierungen, deren Quellcode kostenlos zur Verfügung steht. Sie müssen sich diesen Code besorgen und selbst nach Windows portieren.

RCS ist ein weiteres verbreitetes Versionskontrollsystem. Dieses System verwendet Suffixe für Dateien statt Präfixe (wie SCCS). Dadurch wird das Erstellen von impliziten Regeln für **make** erheblich vereinfacht. RCS ist in den UNIX-Umgebungsprodukten MKS, Interix und Cygwin enthalten.

Darüber hinaus gibt es weitere Quellcodekontrolltools wie Perforce und ClearCase. Die Standardregeln von **make** auf UNIX-Systemen berücksichtigen diese Systeme nicht. Wenn bereits Regeln für diese Systeme vorhanden sind, sollte eine Migration der Regeln nach Windows in allen UNIX-Umgebungen möglich sein.

Dynamische Makros in den Listen vorausgesetzter Dateien

Normalerweise können in der Liste der vorausgesetzten Dateien keine dynamischen Makros verwendet werden. Sie sind auf die Anweisungen beschränkt. Es gibt einen einzigen Fall, in dem das dynamische Makro **\$@** in der Liste der vorausgesetzten Dateien genutzt werden kann. In diesem Fall muss ein zusätzliches Dollarzeichen (\$) am Anfang des Makros eingefügt werden (**\$\$@**). Das folgende Beispiel zeigt, wie in der Liste der vorausgesetzten Dateien auf den aktuellen Zieldateinamen verwiesen werden kann:

```
Datei1 Datei2 Datei3 : $$@.c
```

ist äquivalent mit

```
Datei1 : Datei1.c
```

```
Datei2 : Datei2.c
```

```
Datei3 : Datei3.c
```

Dieses Feature wird in den Solaris-, Interix- und MKS-Versionen von **make** unterstützt, nicht jedoch in **gmake**.

Spezielle Ziele

Alle Versionen von **make** unterstützen einige spezielle Funktionsziele, die von **make** besonders behandelt werden. Einige dieser Ziele sind bekannt und verhalten sich in allen Implementierungen identisch; dazu gehören beispielsweise **.DEFAULT**, **.IGNORE**, **.PRECIOUS** und **.SUFFIXES**.

Jede Implementierung unterstützt jedoch spezielle Ziele, die für die jeweilige Umgebung spezifisch sind. Die Solaris-Version von **make** definiert beispielsweise mehrere Ziele dieser Art, wie zum Beispiel **.INIT**, **.KEEP_STATE**, **.KEEP_STATE_FILE** und **.MAKE_VERSION**. Andere Versionen von **make** verfügen über eine eigene Liste, wie zum Beispiel **DELETE_ON_ERROR** und **.PHONY** in **gmake** oder **.OPTIONAL**, **.BEGIN** und **.END** in den BSD-Versionen von **make**.

In der Tabelle 4.3 sind einige proprietäre spezielle Ziele aufgeführt.

Tabelle 4.3: Proprietäre spezielle Ziele

Solaris	Interix	Gmake	MKS
.INIT	.BEGIN	—	—
.DONE	.END	—	—
(verwendet VPATH-Makro)	.PATH	(verwendet VPATH-Makro)	.SOURCE
.KEEP_STATE	—	—	—
.KEEP_STATE_FILE	—	—	—
.SCCS_GET	—	—	—

Übersetzen von Compileroptionen

Der Befehl **cc** (bzw. **gcc**) bildet die Schnittstelle des Programmierers zum Compiler. Dabei kann es sich um das eigentliche Compilerprogramm oder um einen Wrapper bzw. ein Front-End für den Compiler handeln. Jeder Compiler akzeptiert unterschiedliche Befehlszeilenoptionen. Um diese Optionen konvertieren zu können, müssen Sie die Funktion der verschiedenen Optionen und die entsprechenden Optionen in der ausgewählten Umgebung kennen (sofern äquivalente Optionen vorhanden sind). Der **cc**-Compiler von UNIX unterstützt viele Optionen. Die meisten davon sind auf zahlreichen UNIX-Plattformen identisch. Jede Plattform verfügt jedoch über eigene abweichende Optionen. In Tabelle 4.5 sind einige der gebräuchlicheren Optionen aufgeführt, die beim Aufruf des C-Compilers unter UNIX verwendet werden können, sowie die am ehesten entsprechenden Optionen (sofern vorhanden) der verschiedenen C-Compilerprogramme unter Windows.

Wenn der Buildprozess Compileroptionen erfordert, die nicht unterstützt werden, müssen Sie Problemumgehungen suchen. Sie können das MKS-Dienstprogramm **cc** und das Interix-Programm **wcc** direkt bearbeiten, da es sich um Skripts handelt, die ein Front-End zum Microsoft C-Compiler bilden. Ermitteln Sie aus den Microsoft C-Compileroptionen die am besten geeignete Option, und fügen Sie in das **cc**- oder **wcc**-Skript die Anweisungen ein, die diese Funktionalität bereitstellen.

Die verschiedenen Compiler befinden sich in den folgenden Verzeichnissen:

Tabelle 4.4: Compilerverzeichnisse

Compiler	Verzeichnis
MKS Toolkit for Developers	\$ROOTDIR/etc/compiler.ccg
MKS Toolkit for Enterprise Developers	\$ROOTDIR/etc/nutccg/cc.ccg
SFU Interix	wcc (von http://www.interopsystems.com/)
Cygwin-Tools	/bin/gcc

Der Unterschied der beiden MKS Toolkit-Compilerskripts besteht darin, dass **etc/nutccg/cc.ccg** die MKS NuTcracker-APIs für UNIX-Portierungen unterstützt. Mit dem Skript **etc/compiler.ccg** werden Anwendungen nur mit den in Windows bereitgestellten Bibliotheken und Funktionen erstellt.

Tabelle 4.5 enthält viele der gemeinsamen C-Compileroptionen der verschiedenen Compiler.

Tabelle 4.5: Gemeinsame Compileroptionen

Beschreibung	Solaris	gcc	Interix-wcc	MKS-cc
Kompilieren, aber nicht linken	-c	-c	-c	-c
Keine Kommentare bei Preprocessing entfernen	-C	-C	-C	-C
Makro mit der Definition <i>Variable</i> definieren	-DVariable	-DVariable	-DVariable	-DVariable
Nur Preprocessing	-E	-E	-E	-E
Include-Verzeichnis <i>Include-Verz.</i> angeben (großes I)	-I Include-Verz.	-I Include-Verz.	-I Include-Verz.	-I Include-Verz.
Bibliothek <i>Bibliothek</i> angeben (kleines L)	-I Bibliothek	-I Bibliothek	-I Bibliothek	-I Bibliothek
Bibliotheken in <i>Bibliotheksverzeichnis</i> suchen	-L Bibliotheksverz.	-L Bibliotheksverz.	-L Bibliotheksverz.	-L Bibliotheksverz.
Optimieren	-O	-O	-O	-O
Resultierende Datei <i>Ausgabedatei</i> nennen	-o Ausgabedatei	-o Ausgabedatei	-o Ausgabedatei	-o Ausgabedatei
Nur Preprocessing und Ausgabe in Datei mit der Erweiterung <i>.i</i> speichern	-P	<nicht verfügbar>	-P	-P
Ausführbare Datei ohne Symboltabellen erzeugen	-s	-s	-s	-s
In Assemblercode kompilieren	-S	-S	-S	-S
<i>Argument</i> an Linker oder Compiler bzw. an Komponente <i>Komponente</i> übergeben	-WKomponente,Argument	-WKomponente,Argument	-WArgument <Übergabe an Compiler> -YArgument <Übergabe an Linker>	-W/Argument
Im strikt ANSI-konformen Modus arbeiten	-Xc	-ansi	-Xc	-Xc
Im ANSI-Modus plus Erweiterungen arbeiten (Standardmodus)	-Xa	<nicht verfügbar>	-Xa	-Xa
Im ANSI- und K&R-Modus arbeiten	-Xs	<nicht verfügbar>	<nicht verfügbar>	-Xs
K&R-Modus	-Xt	<nicht verfügbar>	<nicht verfügbar>	-Xt

Bei der MKS-Version von **cc** gibt es ein grundlegendes Problem hinsichtlich der Option **-I**: Die Standardversionen von **-Il** und **-ly** führen zu einem Fehler, weil die Option den Compiler anweist,

Funktionen aus den Unterstützungsbibliotheken **lex** und **yacc** einzubeziehen. Wenn diese Optionen im MKS-Befehl **cc** verwendet werden, wird der folgende Fehler angezeigt:

```
Warning: Could not locate -ll; assuming „l.lib“
```

```
LINK : fatal error LNK1181: cannot open input file l.lib'
```

Das Problem scheint darin zu liegen, dass die Bibliotheken für **lex** und **yacc** mit falschen Namen installiert wurden (als **lex.lib** und **yacc.lib** in **\$ROOTDIR/lib** anstatt als **l.lib** und **y.lib**). Sie können dieses Problem beheben, indem Sie **lex.lib** in **l.lib** und **yacc.lib** in **y.lib** kopieren.

Linker

Mit dem Linker werden die Objektdateien zusammengeführt und die ausführbaren Dateien bzw. die gemeinsamen Bibliotheken erstellt. Die Symbolnamen und Speicherverweise in den Objektdateien werden aufgelöst, so dass eine vollständige ausführbare Datei entsteht. Dieses Dienstprogramm heißt unter UNIX üblicherweise **ld**, unter Windows **link.exe**.

Ausführbare Dateien bzw. gemeinsame Bibliotheken werden normalerweise mit dem Compiler erstellt. Der Compiler führt den Linker automatisch in geeigneter Weise aus. Er unterstützt in der Regel eine Befehlszeilenoption, mit der linkerspezifische Optionen angegeben werden können, die beim Aufruf des Linkers übergeben werden. Bisweilen soll der Linker jedoch direkt aufgerufen werden. In diesem Fall muss festgestellt werden, warum dies unter UNIX erfolgt ist, ob es beim Erstellen des Builds unter Windows weiterhin erforderlich ist und ob gegebenenfalls die Befehlszeilenoptionen für den Linker geändert werden müssen. Wie die Compiler unterstützen die Linker auf den verschiedenen Plattformen unterschiedliche Befehlszeilenoptionen.

Tabelle 4.6: Gebräuchliche Linkeroptionen

Beschreibung	Solaris	ld in GNU	link.exe in Windows	ld in MKS
Den ersten Einstiegspunkt auf <i>Einstiegspunkt</i> festlegen	-e <i>Einstiegspunkt</i>	-e <i>Einstiegspunkt</i>	-entry: <i>Einstiegspunkt</i>	-e <i>Einstiegspunkt</i>
<i>Verzeichnis</i> zu Bibliothekssuchverzeichnissen hinzufügen	-L <i>Verzeichnis</i>	-L <i>Verzeichnis</i>	<nicht verfügbar>	-L <i>Verzeichnis</i>
Bibliothek <i>Bibliothek</i> in Linkvorgang einbeziehen	-l <i>Bibliothek</i>	-l <i>Bibliothek</i>	<Angabe von libx.lib als Operand erforderlich>	-l <i>Bibliothek</i> [sucht libx.a , dann x.lib]
Adresszuordnung in <i>Zuordnungsdatei</i> erstellen	-M <i>Zuordnungsdatei</i>	-M <Ausgabe an stdout >	- map: <i>Zuordnungsdatei</i>	- W/map: <i>Zuordnungsdatei</i>
Resultierende Datei <i>Ausgabedatei</i> nennen	-o <i>Ausgabedatei</i>	-o <i>Ausgabedatei</i>	-out: <i>Ausgabedatei</i>	-o <i>Ausgabedatei</i>
Debugsymbole entfernen	-s	-s	-debug:none	-s

Dynamische und statische Bibliotheken

Beim Erstellen von ausführbaren Programmen werden normalerweise einige Bibliotheken mit den Objektdateien verknüpft. Es gibt zwei Arten von Bibliotheken: statische und dynamische. Statische Bibliotheken werden verwendet, um eine eigenständige ausführbare Datei zu erstellen, die nicht von anderen Bibliotheken abhängig ist. Der gesamte ausführbare Code ist in einer einzigen Datei enthalten. Bei dynamischen Verknüpfungen ist es zur Ausführung der entsprechenden Datei erforderlich, dass die zugeordneten gemeinsamen Dateien vorhanden sind. Diese gemeinsamen Bibliotheken können von vielen verschiedenen ausführbaren Programmen verwendet werden. Die ausführbare Datei ist in diesem Fall nicht so groß wie die statische Version.

Die übliche Erweiterung für gemeinsame Bibliotheken unter UNIX ist **.so**, unter Windows **.dll**. Die Versions- und Identifikationseigenschaften sind unterschiedlich. Diese Unterschiede können aber im Buildsystem verborgen sein.

Migrieren von Shellskripts

Bei der Verwendung von Shellskripts im Buildprozess müssen Sie eventuell mit Problemen rechnen, die bei der Ausführung unter Windows auftreten können. Die meisten Shellskripts werden aber für die Bourne-Shell oder für eine POSIX.2-Shell wie die Korn-Shell oder **bash** geschrieben. Das bedeutet, dass die Skripts selbst selten ein Migrationsproblem darstellen. Die meisten Probleme entstehen aufgrund von Unterschieden im Betriebssystem, wie zum Beispiel Benennungssyntax der Dateisysteme, und bei Befehlszeilenoptionen, die von den anderen Dienstprogrammen unterstützt werden.

Eines der häufigeren Probleme bildet die Verwendung von absoluten UNIX-Pfadnamen wie **/bin/cp** oder **/tmp**. Einige davon, etwa **/tmp**, verarbeitet die UNIX-Umgebung automatisch, andere jedoch nicht. Allein die MKS-Umgebung unterstützt kein Dateisystem mit einem einzigen Stammverzeichnis (**/**), in dem dieses immer auf ein konstantes Verzeichnis im Dateisystem verweist. Die MKS-Umgebung verfügt über mehrere Stammverzeichnisse, jeweils eines für jedes bereitgestellte Dateisystem (d. h. Laufwerkbuchstabe), da Windows auf diese Weise arbeitet. Wenn diese Pfadnamen in den Skripten nicht geändert werden sollen, können symbolische Verknüpfungen zu den Verzeichnissen erstellen werden, so dass die gebräuchlichen Verzeichnisse auf die entsprechenden Pendanten in **\$ROOTDIR** verweisen (genauso wie **/bin** und **/usr/bin** auf **\$ROOTDIR/mksnt** zeigen).

Migrieren sonstiger Befehle

Während eines Buildprozesses werden viele UNIX-Befehle aufgerufen, entweder in den Anweisungen für die Zieldateien in den Makefiles oder in Shellskripten, die während des Buildzyklus aufgerufen werden. Ein Bestandteil des Migrationsprozesses ist es, sicherzustellen, dass diese Befehle auf der Windows-Plattform weiterhin funktionieren. Die Informationen in den folgenden Abschnitten weisen auf verschiedene bekannte Probleme mit einigen der gebräuchlicheren Dienstprogramme hin, die in einem Buildprozess verwendet werden.

lex

Bei dem Dienstprogramm **lex** handelt es sich um ein Tool zum Erstellen von C-Programmen, die eine lexikalische Verarbeitung oder Analyse von Zeicheneingaben durchführen können. Es wird hauptsächlich als Schnittstelle zum Dienstprogramm **yacc** verwendet. **lex** liest eine Reihe von Regeln mit entsprechenden Aktionen ein und erzeugt ein C-Programm, mit dem diese Regeln ausgeführt werden können.

Es gibt mehrere verbreitete Implementierungen von **lex**: **flex** aus GNU, **MKS-lex** sowie das ursprüngliche **AT&T-lex**. Viele kommerzielle UNIX-Implementierungen basieren auf der AT&T-Version, die Interix-Implementierung auf GNU-**flex**. Diese Versionen sind weitgehend kompatibel miteinander, so dass das Portieren einer Datei mit **lex**-Regeln relativ einfach ist und lediglich wenige Änderungen erforderlich sind.

Informationen zu den unterschiedlichen Spezifikationen von GNU-**flex**, AT&T-**lex**, MKS-**lex** und POSIX-**lex** finden Sie in den folgenden englischsprachigen Ressourcen:

http://www.gnu.org/software/flex/manual/html_mono/flex.html

http://www.mkssoftware.com/docs/wp/wp_lyuse.asp

yacc

Bei dem Dienstprogramm **yacc** handelt es sich um einen Parser-Generator. Es erstellt ein C-Programm, mit dem die Eingabe basierend auf bestimmten Regeln und Spezifikationen (der so genannten Grammatik) analysiert werden kann. Der Benutzer gibt einige Aktionen ein, die diesen Regeln entsprechen. Diese Aktionen werden aufgerufen, wenn der Parser eine Übereinstimmung feststellt. Mit dem Dienstprogramm **yacc** werden auf diese Weise viele Sprachparser entwickelt, von einfachen Tischrechnern bis hin zu komplexen Programmiersprachen. **yacc** wird normalerweise zusammen mit dem Dienstprogramm **lex** verwendet.

Es gibt mehrere verbreitete Implementierungen des Dienstprogramms **yacc**: Versionen, die auf dem ursprünglichen AT&T-Code basieren, die öffentliche BSD-Version der University of Berkeley und die von GNU (**Bison**). Die Interix-Version basiert auf der BSD-Implementierung und die MKS-Version auf Code von SCO, der vom ursprünglichen AT&T-Code abgeleitet wurde.

Den Implementierungen aller dieser Versionen liegt das Verhalten der ursprünglichen AT&T UNIX-Implementierung zugrunde. Die Migration der **yacc**-Spezifikationsdatei zwischen den Versionen ist daher normalerweise trivial. Die häufigsten Probleme sind auf interne Einschränkungen zurückzuführen, wie zum Beispiel Puffergrößen und Speichergrenzen. In den meisten Fällen erweist sich die GNU-Version **Bison** als die vielseitigste und ist weniger anfällig für diese Probleme.

awk

Bei dem Dienstprogramm **awk** handelt es sich um eine Sprache für Zeichenfolgemanipulation und Berichterzeugung. Es wird für komplexe Texttransformationen basierend auf Musterübereinstimmungen verwendet.

Die Versionen von **awk** sind auf allen Plattformen einheitlich. Obwohl die ältesten UNIX-Systeme zwischen dem „alten“ **awk** und dem „neuen“ **awk (nawk)** unterscheiden, verwenden moderne UNIX-Systeme ausschließlich die neue Variante. Die einzige Implementierung, die **awk** wesentlich erweitert, ist GNU-**awk** in Cygwin (**gawk**). **gawk** ist jedoch für Windows verfügbar. Wenn Sie also **gawk** unter UNIX verwendet haben, können Sie auch unter Windows problemlos **gawk** einsetzen.

Ein gelegentliches Kompatibilitätsproblem ist die Iterationsreihenfolge für ein Array in einer **for**-Anweisung. Die Iterationsreihenfolge kann von der jeweiligen Implementierung abhängen, weil keine Garantie dafür besteht, in welcher Reihenfolge das Programm das Array durchläuft. In **awk** sind alle Arrays assoziativ. Es gibt also keine implizite Anordnung nach Index. Die **for**-Anweisung kann die Elemente in beliebiger Reihenfolge durchlaufen.

cp

Bei dem Dienstprogramm **cp** handelt es sich um das Standardtool zum Kopieren von Dateien unter UNIX. Es gibt zwei Optionen, **-r** und **-R**, bei denen der Kopiervorgang rekursiv erfolgt (einschließlich Verzeichnissen und speziellen Dateien). Die Optionen können sich hinsichtlich der Behandlung von speziellen UNIX-Dateien unterscheiden. Das Verhalten von **-R** ist durch POSIX.2 geregelt, das Verhalten von **-r** aber nicht. Untersuchen Sie die Shellskripts und Makefiles nach Vorkommen von **cp -r**, und stellen Sie fest, wie die speziellen Dateien behandelt werden sollen.

chmod

Das Dienstprogramm **chmod** ändert die UNIX-Berechtigungsbits für eine Datei. UNIX-Berechtigungen basieren auf drei Bitgruppen, Windows-Berechtigungen hingegen auf Zugriffssteuerungslisten, einem flexibleren, aber komplizierteren Mechanismus. Ausführliche Informationen zu Berechtigungen finden Sie im *UNIX Application Migration Guide* (englischsprachig).

Da Windows über einen wesentlich komplexeren Zugriffssteuerungsmechanismus verfügt, ist in den meisten UNIX-Umgebungen unter Windows keine präzise Zuordnung zwischen den UNIX- und Windows-Mechanismen implementiert. Dies ist lediglich bei Interix der Fall. Wenn Sie eine andere Umgebung als Interix verwenden, müssen Sie vorsichtig sein, wenn UNIX-Befehle verwendet werden, die Dateiberechtigungsbits ändern, da unter Umständen nicht das erwartete Ergebnis erzielt wird.

diff

Das Dienstprogramm **diff** vergleicht zwei Textdateien und stellt eine Liste mit den Unterschieden bereit. Diese Unterschiede werden normalerweise in einer Form dargestellt, die in ein Skript eingefügt werden kann, um die eine Textdatei in die andere umzuwandeln.

Bisweilen verwenden Buildtools oder Entwickler das Dienstprogramm **diff**, um festzustellen, ob bestimmte Dateien identisch oder verschieden sind. Die meisten Implementierungen von **diff** können Text- und Binärdateien problemlos vergleichen. In der MKS Toolkit-Version ist es jedoch nicht möglich, mit **diff** Binärdateien abzugleichen. **diff** erzeugt für diese dann eine Fehlermeldung. Wenn bekannt ist, dass Binärdateien verglichen werden, können Sie dieses Problem umgehen, indem Sie das Dienstprogramm **cmp** verwenden.

In

Das Dienstprogramm **In** erstellt neue Dateinamen und verknüpft sie mit einer vorhandenen Datei. Es wird verwendet, um das Kopieren von Dateien zu vermeiden. Es gibt zwei Arten von Verknüpfungen: feste und symbolische. Alle **In**-Dienstprogramme in diesen Systemen können feste Verknüpfungen erstellen, wenn ein Windows-NTFS-Dateisystem verwendet wird.

Bei Interix kann das Dienstprogramm **In** symbolische Links zu Dateien und Verzeichnissen erstellen. Diese Verknüpfungen werden allerdings nur in der Interix-Umgebung als solche erkannt. Die symbolisch verknüpften Dateien werden beim Zugriff über die Windows-Umgebung als Systemdatendateien angezeigt.

Das MKS-Dienstprogramm **In** kann über Windows-Verbindungspunkte symbolische Verknüpfungen zu einem Verzeichnis erstellen. Entsprechende Verknüpfungen zu Dateien sind hingegen nicht möglich.

Das Cygwin-Dienstprogramm **In** unterstützt die Option **-s**, um symbolische Verknüpfungen zu erstellen. Dazu wird eine LNK-Datei (Windows-Verknüpfungsdatei) erstellt. Es handelt sich dabei nicht um eine echte symbolische Verknüpfung, sondern um ein Feature der Windows-Shell, das nur mit Windows-Dienstprogrammen (wie Explorer) funktioniert, die diese Funktion erkennen. Für viele andere Windows- und Interix-Dienstprogramme ist die symbolische Verknüpfung eine normale Datei, die Binärdaten enthält.

sed

Bei dem Befehl **sed** handelt es sich um einen Streameditor, mit dem Textstreams geändert werden können. Er wird häufig verwendet, um die Ausgabe so zu transformieren, dass eine neue Befehlsdatei oder ein neues Makefile entsteht.

In Bezug auf **sed** sind zwei Migrationsprobleme weit verbreitet. Das erste Problem besteht darin, dass ältere Versionen von **sed** oftmals nur über eine Teilmenge der regulären Ausdrücke verfügen und die implementierte Teilmenge vom jeweiligen System abhängt. Die **sed**-Versionen von MKS, Interix und Solaris sollen POSIX.2-kompatibel sein, so dass reguläre Ausdrücke portierbar sein müssten. Die Cygwin-Version von **sed** soll auch POSIX.2-kompatibel sein. Es gibt aber einige Inkompatibilitäten, die auf der Manpage zu **sed(1)** beschrieben sind. Wenn Sie keine POSIX.2-kompatible Version von **sed** verwenden, müssen Sie die verwendeten regulären Ausdrücke untersuchen und die gegebenenfalls erforderlichen Änderungen ermitteln. Die Zeichen Punkt (.) und Sternchen (*) für reguläre Ausdrücke werden immer unterstützt. Die Interpretation von Metazeichen in Klammerausdrücken (wie **[A-z*]**) kann jedoch bei Nicht-POSIX-Implementierungen variieren.

Das zweite Problem ist eine Inkompatibilität bei der Ausgabe. Der Befehl **sed** verfügt über eine automatische Ausgabefunktion, die automatisch die verglichenen Zeilen ausgibt. Darüber hinaus gibt es den Befehl **p**, um die Ergebnisse einer Operation auszugeben. Bei einigen Implementierungen wird die Zeile doppelt ausgegeben, wenn die automatische Ausgabe nicht deaktiviert und der Befehl **p** verwendet wird. Bei anderen wird nur eine Zeile ausgegeben. Beides ist gemäß POSIX zulässig, also kein Fehler. Wenn in den **sed**-Skripts beide Ausgabeverhalten vorkommen, sind sie unter Umständen nicht portierbar. Sie müssen neu erstellt werden, so dass die Option **-n** verwendet und explizit das gewünschte Ergebnis ausgegeben wird.

Erstellen oder Migrieren von Systemtests

In dieser Phase müssen Systemtests erstellt werden, die mit den zuvor erstellten Makefiles ausgeführt werden.

Es ist möglich, dass bereits Tests vorhanden sind, mit denen das UNIX-Buildsystem überprüft wurde. Sie sollten dann nach Windows migriert und als Grundlage für das Testsystem verwendet werden. Buildsystemtests sind jedoch selten, so dass sie wahrscheinlich erstellt werden müssen. Die Testspezifikation wird hier implementiert.

Mit den anfänglichen Verifikationstests ermitteln Sie die zu verwendende UNIX-Umgebung. Nutzen Sie die Ergebnisse als Grundlage, und entwickeln Sie daraufhin die Tests. Diese werden regelmäßig ausgeführt, wenn das Buildsystem in die Stabilisierungsphase übergeht.

Integrieren der Lösung

In dieser Phase muss unbedingt bedacht werden, dass eine Umgebung erstellt wird, die auf den Computern der Endbenutzer installiert wird. Faktoren wie Umgebungsvariablen können von Bedeutung sein. Diese Faktoren müssen von der Benutzerrolle erfasst werden, um das Buildsystem für die Endbenutzer zu dokumentieren.

Einige Komponenten können nicht isoliert getestet werden: Es ist nicht aufwändig, einen Stub des Befehls **cp** bereitzustellen, damit Dateien nicht verschoben werden. Der Aufwand, den Befehl **cp** zu ersetzen, ist größer als der Aufwand, Stubdateien für Tests bereitzustellen. Elemente wie Shellskripts beinhalten und testen andere Komponenten automatisch. Schließlich müssen alle Teile zusammengefügt werden.

Buildsysteme sind so beschaffen, dass die Entwickler unter Umständen gleichzeitig an den unterschiedlichen Komponenten arbeiten: Zunächst werden die Makefiles geändert, um die richtigen Datei- und Pfadnamen sowie die zutreffenden Compileroptionen zu ermitteln. Anschließend wird mit einem schnellen **make**-Befehl überprüft, ob keine Probleme vorliegen. Daraufhin werden einige Zeilen der Anweisung korrigiert und das Shellskript bearbeitet, das beim Erstellen der Zieldateien aufgerufen wird. Schließlich wird das Makefile fertig gestellt.

Testen der Lösung

Die ersten Tests der Komponenten werden auf den Entwicklungssystemen durchgeführt. Sie werden in die Beispielumgebung verschoben und erneut getestet. Das Verschieben der Buildkomponenten in die Test- und Beispielumgebung ist ein wichtiger Schritt, um die Funktionsfähigkeit des Buildsystems zu erreichen. Genauso, wie es gelegentlich notwendig ist, alle Zwischendateien zu entfernen und eine Anwendung nur aus den Quelldateien zu erstellen, ist es erforderlich, die Zwischendateien in der Beispielumgebung zu löschen, die aktuelle Version des Buildsystems neu zu installieren und den Vorgang zu wiederholen.

Am Ende dieser Phase sollte das Buildmigrationsteam Folgendes erreicht haben:

- Ein ausführbares Buildsystem, auch wenn noch Probleme vorhanden sein können.
- Eine vorläufige Dokumentation, die das Installieren und Verwenden des Buildsystems beschreibt, wie zum Beispiel Dateinamenzuordnungen und die Verwendung der UNIX-Portabilitätsumgebung.

Eine Reihe von Systemtests, mit denen das Verhalten des Systems vermessen wird. Die Messungen wurden im Testplan definiert und hängen von den jeweiligen Anforderungen des Unternehmens ab.

[Feedback an Microsoft](#)

© 2004 Microsoft Corporation. Alle Rechte vorbehalten.