Hochportable Software

Jörg Schilling

BerliOS, FhG-Fokus

ABSTRACT

Eine Beschreibung wie Software mit vertretbarem Arbeitsaufwand auf vielen unterschiedlichen Plattformen lauffähig wird.

1. Vorwort

Das *Schily* Makefilesystem ist nun seit über 5 Jahren bewährt und hat die systematische Portierung von vielen Programmen, die insgesamt einen großen Bereich unterschiedlicher Anwendungsgebiete andecken, erheblich erleichtert. Seit Januar 2001 ist sogar ein Funktionsumfang erreicht, mit dem das *Schily* Makefilesystem mehr unterschiedliche Plattformen unterstützt und eine merklich höhere und vor allem auch einfachere Portabilität garantieren kann, als bei der ausschließlichen Verwendung von Portabilitätshilfmitteln der FSF[1] erreichbar ist.

Bei der Erstellung dieser Dokumentation fiel jedoch auf, daß speziell die Struktur der Includedateien, die zur Systematisierung und Vereinfachung der Portierung der Programme zur Verfügung gestellt werden, zu komplex in der Anwendung war. Es gab zwar Regeln für die korrekte Verwendung, doch waren diese Regeln so komplex, daß sie nur schwer in einem Papier wie diesem vermittelbar gewesen wären. Aus diesem Grund wurde die Struktur dieser Includedateien noch einmal komplett überarbeitet und so eine weitere erhebliche Vereinfachung des Portierungsprozesses unter Verwendung der hier beschriebenen Methode erreicht.

Es ist nun möglich, einen Satz von wenigen verhälnismäßig einfachen Regeln aufzustellen die, wenn man sie beachtet, eine Portabilität im Bereich der POSIX[2] konformen Betriebssysteme garantieren.

---

[1] Free Software Foundation

[2] Siehe **http://www.opengroup.org**

Die Erlangung einer möglichst großen Portabilität von freier Software erscheint wichtig, um sicherzustellen daß Open Source Software auf einer großen Anzahl von unterschiedlichen Plattformen lauffähig ist. Da portable Open Source Software es den Anwendern ermöglicht auf nahezu jedem Betriebssystem freie Software einzusetzen, erlaubt dies einen „weichen Umstieg“ auf die Open Source Alternative. Portable Open Source Software stellt sicher, daß das **Diktat unportabler „unfreier“ Software** einer Weltmarktbeherrschenden Softwarefirma nicht durch das **Diktat unportabler „freier“ Software** ersetzt wird, die den Anwendern durch ihre mangelde Portabilität letztlich auch nicht die Freiheit eines Systemwechsels einräumt. Wäre freie Software nicht portabel, würde ihr einziger Vorteil gegenüber proprietärer Software darin liegen, daß sie kostenlos ist.

2. Zum Verständnis von Portabilität

Zunächst sollten wir und damit beschäftigen, was **Portabilität** eigentlich bedeutet und dann die Probleme bei der Erstellung portabler Software diskutieren. Danach können wir nach den besten Lösungen für die erkannten Probleme suchen.

2.1. Was ist Portabilität?

Ein Programm kann dann als **portabel** angesehen werden, wenn es auf einer ausreichend großen Anzahl von unterschiedlichen Plattformen übersetzbar ist und das Ergebnis in befriedigender Weise d.h. möglichst mit identischem Funktionsumfang und mit identischer Funktionalität auf allen unterstützten Plattformen lauffähig ist.

Dieses Ziel läßt sich gewöhnlich leichter erreichen wenn das betreffende Programm nicht sehr stark von bestimmter Hardware oder Eigenschaften eines Betriebssystems abhängt. Ein Programm welches auf nicht stark verbreitete Merkmale eines Betriebssystems aufbaut, oder in direkter Weise versucht auf Hardware zuzugreifen, wird nur mit größerem Aufwand auf viele Plattformen zu portieren sein.

2.2. Was ist eine Plattform

Da die Portabilität primär im Zusammenhang mit zu unterstützenden unterschiedlichen Plattformen gesehen wird, ist es nötig zu definieren was eine Plattform ist.

Eine bestimmte Plattform wird durch eine Kombination aus Hardware und Software definiert. Im Allgemeinen ist für die Hardware im Wesentlichen die CPU entscheidend. Falls das zu portierende Programm direkt auf Peripherie zugreift, ist natürlich auch diese Peripherie von Einfluß auf die Portabilität. Bei der Software die die Portabilität eines Programms bestimmt ist im Wesentlichen das verwendete Betriebssystem (d.h. der Betriebssystemtyp z.B. Solaris oder Linux) sowie die verwendete Version des Betriebssystem entscheidend.

3. Warum sind Programme nicht portabel

Um die Gründe für mangelnde Portabilität zu verstehen, muß man sich zunächst mit den Randbedingungen für die Kompilation und den Laufzeiteigenschaften der unterschiedlichen Plattformen beschäftigen. Diese Randbedingungen sind im Einzelnen:

3.1. Das Betriebssystem

Jedes Betriebssystem hat einen unterschiedlichen Satz von Leistungsmerkmalen. Programme die einen großen Satz von Leistungsmerkmalen für ihre Funktion benötigen sind tendenziell weniger portabel, denn es ist denkbar, daß ein bestimmtes Leistungsmerkmal auf einem der gewünschten Zielplattformen nicht verfügbar ist. Es ist daher wichtig von Anfang an bei der Konzeption eines Programmsystems zu planen welche Leistungsmerkmale des Betriebssystems genutzt werden soll. Als ein Beispiel kann hier die Verwendung von *System V Semaphoren* kann genannt werden. *System V Semaphoren* können Probleme mit der Portabilität bringen, eine Verwendung von *UNIX Pipes* kann in vielen Fällen die Semaphoren ersetzen und bewirkt keine Einschränkung der Portabilität, denn Pipes sind sogar unter Win32 verfügbar.

Häufig ist es auch möglich die gleiche Funtionalität durch die Verwendung anderer - aber weiter verbreiteter Leistungsmerkmale - zu erreichen. Auch kann ein Programm eine höhere Portabilität erreichen, wenn bei der Planung darauf geachtet wird daß unterschiedliche Leistungsmerkmale zur Implementierung bestimmter Programmeigenschaften eingesetzt werden können. Dies ist z.B. der Fall wenn man *Shared Memory* entweder durch das Leistungsmerkmal *System V shared memory* oder durch das Leistungsmerkmal *Mapped anonymous memory* abbildet, je nachdem welches der beiden Leistungsmerkmale das aktuelle Betriebssystem zur Verfügung stellt. Ein geeignetes Programm zu Testen auf das Vorhandensein bestimmter Leistungsmerkmale ist **GNU Autoconf**[3].

3.2. Der Prozessor

Auch ein anderer Prozessor kann bewirken daß ein Programm nicht mehr ordnungsgemäß funktioniert. Häufigste Ursachen hierfür sind unterschiedliche Byteorder sowie andere Anforderungen des Prozessors an das *Alignment* von Variablen und Datenstrukturen. Hier zeigt es sich, daß Intel basierte Entwicklungssysteme weniger zur Entwicklung geeignet sind, da Intel Prozessoren kein Alignment vorschreiben - ein *long int* wird selbst auf einer ungeraden Adresse akzeptiert. Wenn man sich bei der Entwicklung der Programme die Alignmentproblems nicht bewußt ist, dann bemerkt man es frühestens wenn das Programm das erste mal auf einem anderen Prozessor getestet wird. Während das Alignmentproblem sich jedoch dadurch lösen läßt, daß man Strukturen generell so ausrichtet, daß von den ungünstigsten Voraussetzungen ausgegangen wird (d.h. jedes Element einer Struktur bis zu der Größe eines *doubles* fängt auf einer Adresse an, die sich durch die Größe des Objekts teilen läßt), ist das Problem der unterschiedlichen Byteorder der Prozessoren nur durch Systematische Abstraktion von der Byteorder des Prozessors zu lösen. Zum Erkennen der auf dem aktuellen Prozessor gültigen Byteorder kann man bei einigen Betriebssystemen entsprechende Systemincludedateien verwenden oder man nimmt das universelle **GNU Autoconf**. Für das Alignment gibt es im *Schily* Makefilesystem eine entsprechende Includedatei mit den dafür notwendigen Definitionen.

3.3. Die Systembibliotheken

Neben dem Betriebssystem können auch die zum System gehörenden Standardbibliotheken einen unterschiedliche Funktionsumfang zur Verfügung stellen. Meistens handelt es sich hierbei um Probleme durch unterschiedliche Herangehensweisen bei der Implementierung dieser Bibliotheken aus der Zeit vor der Standardisierung durch POSIX-1003.1-1990, UNIX-98[4] bzw. POSIX-1003.1-2001. Auch hier ist ein geeignetes Werkzeug zur Abstraktion von den Unterschieden das Programm **GNU Autoconf**.

[3] Siehe **http://www.gnu.org/software/autoconf/autoconf.html**

[4] Siehe **http://www.unix-systems.org/unix98.html**

3.4. Die Systemincludefiles

Ähnlich wie bei den Systembibliotheken gibt es auch bei den Systemincludefiles Unterschiede die hauptsächlich aus der Zeit vor der Standardisierung stammen. Einige Betriebssysteme haben zusätzlich mehr oder weniger absichtliche Abweichungen vom Standard. Auch hier ist ein geeignetes Werkzeug zur Abstraktion von den Unterschieden das Programm **GNU Autoconf**.

3.5. Eigenschaften des make Programms

Auch die Eigenschaften des üblicherweise zum Einleiten und zur Kontrolle der Kompilation verwendeten **make** Programms unterscheiden auf verschiedenen Plattformen. Daher wäre eine denkbare Lösung nur einen Subset der Eigenschaften der auf den diversen Plattformen vorzufindenden **make** Implementierungen zu verwenden.

Doch obwohl es einen POSIX Standard für das **make** Kommando gibt, reichen selbst die in POSIX.1-2001 definierten Eigenschaften des **make** Kommandos nicht aus um eine wirklich portable Kompilationsumgebung unter ausschließlicher Verwendung von im POSIX Standard definierten Eigenschaften des **make** Programms zu implementieren.

Die fortschreitende Standardisierung bringt offensichtlich noch keine Hilfe, will man nicht unterschiedliche Makefiles für die unterschiedlichen Zielplattformen verwenden. Eine mögliche Lösung für dieses Problem bietet jedoch (wie weiter unten genauer ausgeführt wird) die Konzentration auf wenige **make** Implementierungen die ihrerseits portabel sind.

3.6. Kompiler Eigenschaften

Bei den für die Portabilität relevanten Eigenschaften der Kompiler sind einerseits die unterschiedlichen Optionen und andererseits die unterschiedliche Genauigkeit bei der Implementation des Sprachstandards zu erwähnen.

Die unterschiedlichen Optionen der Kompiler müssen durch eine geeignete Abstraktion der *Makefiles* behandelt werden. Die unterschiedliche Genauigkeit in der Implementation des Sprachstandards stellt ein nicht zu unterschätzendes Problem dar. Viele Programmierer setzen daher auf einen portablen Kompiler und verwenden ausschließlich den **GNU C-Kompiler**.

Dies ist zwar eine naheliegende Idee, eine Ausschließliche Verwendung des **GNU C-Kompilers** während der Entwicklungsphase ist jedoch auch nicht anzuraten falls man wirklich portable Programme erzeugen will. Der GNU Kompiler implementiert diverse Erweiterungen zum C-Sprachstandard die bei anderen Kompilern nicht oder in anderer Weise gefunden werden. Bei Verwendung dieser GNU Kompiler spezifischen Erweiterungen wird erreicht, daß eine Kompilation dann ausschließlich mit Hilfe des GNU Kompilers erfolgen kann. Weiterhin akzeptiert der GNU Kompiler einige nach C-Sprachstandard syntaktisch fehlerhafte Programme ohne auch nur eine Warnung auszugeben.

Will man daher seine Programme wirklich portabel halten, dann sollte man zumindest gelegentlich auch mit anderen Kompilern arbeiten. Kompiler die gute (auf anderen Systemen nicht einstellbare) Fehlermeldungen ausgeben, sind der SunPRO (jetzt Forte) Kompiler, der C-Kompiler von SGI/IRIX und der C-Kompiler der auf Compaq/True64 angefunden wird.

Der SunPRO-Kompiler ist deutlich standardkonformer als der GNU C-Kompiler und gibt gute Warnungen aus falls nichtzulässige Pointerzuweisungen durchgeführt werden. Das können z.B. *signed char ** zu *unsigned char ** Zuweisungen sein, die in Zusammenhang mit Funktionsparametern der POSIX String Funktionen häufig vorkommen und vom GNU C-Kompiler leider nur in der höchten Warnungsstufe, in der typischerweise regelmäßig hunderte von unerwünschten Warnungen ausgegeben werden, angezeigt werden.

Auch der SGI/IRIX C-Kompiler ist standardkonformer als der GNU C-Kompiler und gibt gute Fehlermeldungen aus, wenn eine Zuweisung die eine Verringerung der Genauigkeit bewirkt ohne expliziten Cast durchgeführt wird.

Auch der Der True64 ist standardkonformer als der GNU C-Kompiler und gibt gute Fehlermeldungen aus, wenn durch statische Überprüfung des Kodes Arraygrenzüberschreitungen erkannt werden können.

4. Wie erreicht man Portabilität

4.1. Historische Betrachtungen zu portablen Programmen

Zu den ersten wirklich portablen Programmen gehört das Usenet News System. Die Autoren waren **Mark Horton, Matt Glickman, Rick Adams,** und **Larry Wall.**

Zu den Zeiten als das Usenet System entwickelt wurde (d.h. Anfang der 80'er Jahre) war es üblich #ifdefs die am Plattformnamen orientiert sind (wie z.B. `#if defined(BSD4_2)` oder `#ifdef USG)` zu verwenden um bedingten Code für bestimmte Betriebssyteme zu aktivieren. Es stellte sich jedoch sehr schnell heraus, daß diese Methode nicht mehr sinnvoll anwendbar ist wenn die Komplexizität der Abhängigkeiten steigt.

Daher entstand schon früh die Idee den bedingten Code, der für eine Portierung nötig ist, so einfach wie möglich zu halten. Der einfachste bedingte Code entsteht dann, wenn durch die Bedingung in einem Kodefragment jeweils nur ein einziger Portabilitätsaspekt abgedeckt wird. Es entstand daher die Idee für jede der notwendigen Unterschiede im Code eine eigene Bedingung und ein eigenes #ifdef zu definieren.

Leider gab es in dieser Zeit zu diesem Punkt noch keine Zusammenarbeit der Open Source Software Autoren so daß in den Jahren zwischen 1980 und 1990 jeder der Autoren von freier portabler Software seine eigene Methode zur Konfiguration der Abhängigkeiten entwickelt hatte. Als Ergebnis dieser Parallelentwicklungen war es unmöglich Code eines Autors (obwohl es sich um freie Software handelte) in Programmen anderer Autoren zu verwenden, denn jede Methode implizierte unterschiedliche Randbedingungen zur Erreichung der Portabilität.

4.2. GNU autoconf

GNU autoconf ist der erste systematische Ansatz zur Erlangung von Portabilität der separat von bestehenden Kompilationsumgebungen entwickelt wurde. Die Grundidee hinter autoconf besteht darin, die gesamte Autokonfiguration zu automatisieren. Dieser Ansatz wird im Programm autoconf dadurch realisiert, daß ein automatisch generiertes Shell Script verteilt wird, welches in der Lage ist einzelne Eigenschaften einer Plattform zu testen und dann am Ende der Konfigurationsprozedur eine Datei zu schreiben welche CPP[5] konforme #define Anweisungen enthält. Dieses Shell Script muß allerdings, damit es tatsächlich in der Lage ist die nötigen Tests auf allen Plattformen durchführen zu können, selbst ausreichend portabel sein.

Erste Versionen von GNU autoconf entstanden im Jahre 1989, seit etwa 1995 benutzt der überwiegende Teil der Open Source Software Autoren autoconf zur Elangung der notwendigen Portabilität. Würde es eine brauchbare Dokumentation für autoconf geben und wäre die Hürde die der ersten Verwendung von autoconf entgegensteht nicht so hoch würden vermutlich alle Autoren portabler Software autoconf verwenden.

GNU Autoconf ist vom Grundkonzept eine gute Sache, denn der Wesentliche Ansatz besteht darin, ein Shell Script automatisch zu erzeugen. Dieses Shell Script ist ein Ergebnis einer Kombination von Makros die in der unter UNIX üblichen Makro Sprache **m4** formuliert wurden. Da die einzelnen Makros sehr sauber strukturiert sind und wesentlicher Code jeweils nur einmal existiert ist es möglich ein hochkomplexes Shell Script zu erzeugen, daß nur aus Komponenten

[5] Der C-Präprozessor

zusammengesetzt ist die in ihrer Portabilität ausreichend getestet werden können.

Ein Problem daß der globalen Verwendung von autoconf entgegensteht ist die hohe Anfangskomplexizität, die eine Hürde vor der ersten Verwendung aufbaut. Dieses Problem wird, wie im folgenden ausgeführt wird, durch das *Schily* Makefilesystem gelöst ohne daß dadurch eine Verwendung bei komplexen Programmsystemen verhindert wird.

Ein weiteres zu erwähnendes Problem von autoconf ist, daß es nur etwa zu 50% fertig ist, denn für einen Großteil der häufig in der Praxis vorkommenden Probleme gibt es noch keine Standardtests. Außerdem ist zu erwähnen, daß der Entwurf der dokumentierten Anwendungsvorschläge teilweise in eine ungeeignete Richtung geht. Die offizielle Dokumentation schlägt vor, bestimmte in der Dokumentation angeführte Beispielkodesegmente abzutippen und in die eigenen Programme zu übernehmen. Diese Vorgehensweise ist aber problematisch, denn falls sich herausstellen sollte, daß eines der dokumentierten Kodesegmente fehlerhaft ist, müssen mühselig sämtliche Kopien gesucht und jeweils einzeln korrigiert werden. Das *Schily* Makefilesystem stellt daher Includedateien zur Verfügung, die häufig benötigte Kodefragmente enthalten. Falls sich in diesen Kodefragmenten ein Fehler finden sollte, muß lediglich das betreffende Kodefragment in einer Includedatei korrigiert werden und sämtliche Programme die es benutzen sind damit automatisch korrigiert.

Während autoconf auf optimale Weise eine Includedatei mit Definitionen zur Eigenschaftsorientierten bedingten Kompilation liefert, ist der Ansatz autoconf auch zur Erzeugung bzw. zur Modifikation von Makefiles zu verwenden ungeeignet das Problem optimal zu lösen. Will man eine Kompilationsumgebung erstellen, die die Verwendung des **make** Programms einbezieht, dann macht es keinen Sinn den Anwender zu zwingen vorher eine manuelle Aktion wie das Aufrufen des durch autoconf erzeugten Shell Scripts vorzunehmen denn **make** ist durch Regeldateien so konfigurierbar, daß entsprechende Aktionen automatisiert werden können.

4.3. GNU automake

GNU automake ist definitiv ein Schritt in die falsche Richtung. Eine sinnvolle Vorgehensweise wäre es, wie das *Schily* Makefilesystem es tut, eine Hülle um das autoconf Programm zur Verfügung zu stellen. Doch leider stellt *GNU*automake lediglich einen Versuch dar, der durch autoconf vorgenommenen Modifikation der Makefiles eine weitere Stufe voranzustellen. Wie man sieht, ist schon der Name unglücklich gewählt, denn es handelt sich nicht um ein *automake* Programm, sondern lediglich um einen Makefilegenerator, der schwer überschaubare und damit schwer wartbare Makefiles produziert.

Generell ist zu *GNU*automake zu bemerken, daß es wenig erstrebenswert erscheint eine durch statische Modifikation von Makefiles erzeugte Kompilationsumgebung zu erstellen. Da das **make** Programm es erlaubt eine regelbasierte dynamische Kompilationsumgebung zu verwenden, sollte man bevorzugt eine Kompilationsumgebung erzeugen die sich die Eigenschaften des **make** Programms zu Nutze macht. Das *Schily* Makefilesystem ist, wie weiter unten ausgeführt wird, eine solche Kompilationsumgebung.

5. Generelle Vorgehensweise

Nach Berücksichtigung der oben gemachten Ausführungen erscheint es vernünftig, autoconf dazu zu verwenden um eine automatische Konfiguration der Eigenschaften des C-Kompilers, der Systemincludefiles und der Systembibliotheken durchzuführen. Weiterhin erscheint es sinnvoll eine vorbereitete Testsuite zur Verfügung zu stellen, die alle die Eigenschaften testet, die üblicherweise bei der Programmierung von Interesse sind aber nicht dem autoconf Paket beiliegen. Dadurch wird das Problem, daß die initiale Verwendung von autoconf für die meisten Leute ein unüberwindliches Hinderniss darstellt, vermieden. Durch die Bereitstellung erprobter Tests, die in Erweiterung der in der autoconf Distribution enthaltenen Tests vorhanden sind, wird vermieden daß unerfahrene Programmierer die nötigen Zusatztests in ungeeigneter Weise formulieren.

Wie bereits weiter oben ausgeführt, erscheint es nicht sinnvoll die in autoconf vorhandenen Möglichkeiten Makefiles zu modifizieren auch zu nutzen. Stattdessen erscheint es günstiger die in der Umgebung des **make** Programmes liegenden Notwendigkeiten der Abstraktion von der Zielplattform auf anderem Wege unter Verwendung der Eigenschaften des make Programms zu implementieren.

5.1. Das *Schily* Makefilesystem

Das *Schily* Makefilesystem[6] ist ein integrierter Ansatz die oben ausgeführte Methode der Bereitstellung einer einfach zu bedienenden Portabilitätsumgebung zu implementieren.

[6] Zu finden unter **ftp://ftp.fokus.gmd.de/pub/unix/makefiles/**

5.1.1. Die Struktur des *Schily* Makefilesystem

Das *Schily* Makefilesystem bietet eine integrierte Umgebung, die ausschließlich mit Hilfe des **make** Programms bedient wird. Durch die Verwendung von strukturierten objektorientierten Regeldateien ist eine leichte Wartbarkeit sichergestellt, denn jede Regel existiert nur ein mal im gesamten System.

Vom *Schily* Makefilesystem werden folgende make Programme Unterstützt:

- Das **smake**[7] Programm, es ist das bevorzugte Programm für die Verwendung mit dem *Schily* Makefilesystem denn das *Schily* Makefilesystem wurde und wird unter Verwendung von **smake** entwickelt. **Smake** ist ein hochportables Open Source Programm. Es erfüllt daher die wichtigste Voraussetzung für eine portable Kompilationsumgebung die auf **make** Eigenschaften aufbaut die nicht auf jeder Plattform in der Grundausstattung zu finden sind. Smake wird seit 1984 gepflegt und ist somit älter als gnumake. Smake wird bislang nicht mit dem Ziel entwickelt, daß es zu einem universell verwendbaren make Programm wird. Dies ist für den Einsatz in Verbindung mit dem *Schily* Makefilesyste aber kein Problem, da Portabilität und Benutzbarkeit mit den *Schily* Makefilesystem bislang Vorrang bei der Definition der Entwicklungsziele haben.
- Das **GNU**make Programm. Es hat zwar einige Fehler die dazu führen, daß teilweise verwirrende Fehlermeldungen ausgegeben werden weil gnumake die include Anweisung in Makefiles nicht korrekt behandelt, aber es ist auf vielen Systemen bereits installiert und es erfüllt ansonsten die Mindestanforderungen. Es gibt daher den potentiellen Nutzerns von mit Hilfe des *Schily* Makefilesystems entwickelten Programmsystemen die Möglichkeit sofort eine Kompilation durchzuführen ohne vorher weitere Installationen durchführen zu müssen.
- Das Sun make Programm wie es bei SunOS 4.0 und SunOS 5.x gefunden wird. Es ist zwar nicht portabel aber es ist das einzige sonstige Makeprogramm daß die Mindestkriterien erfüllt.

Es kann durchaus eine Unterstützung für weitere make Programme implementiert werden. Voraussetzung dafür, daß ein make Programm durch das *Schily* Makefilesystem unterstützt werden kann, sind drei Erweiterungen gegenüber den im POSIX.1 Standard aufgelisteten Eigenschaften:

- Das make Programm muß entweder über im Folgenden beschriebende **automake** Eigenschaften verfügen oder in der Lage sein Makro Werte aus dem Ergebnis eines Shell Kommandoaufrufs zu setzen.

---

[7] Unter **ftp://ftp.fokus.gmd.de/pub/unix/smake** ist der Quellcode zu finden.

- Die *include* Anweisung muß implementiert sein. Bei der Bearbeitung der *include* Anweisung müssen Makroexpansionen für die Dateinamen durchgeführt werden.
- Das betreffende make Programm muß sogenannte **Pattern matching Makro Expansionen**[8] durchführen. Damit wird es möglich, Bedingungen zu formulieren und Dateinamen zu generieren die von der Kompilationsumgebung abhängig sind.

Der Grundgedanke, der hinter der Struktur des *Schily* Makefilesystems steckt, besteht darin make Regeln mit Hilfe der **include** Anweisung im make Programm passend zu Plattform und Target einzubinden. Dieser Gedanke ist nicht wirklich neu. Ähnliche Basisansätze finden sich auch seit Februar 1988 in der von Sun für die Kompilation des Betriebssystems SunOS verwendeten Makefiles. Auch die Makefilestruktur wie sie unter BSD seit Juni 1990 verwendet wird verwendet dieses Konzept. Eine ähnliche Methode ist spätestens 1987 im Makesystem von Plan 9 verwendet worden. Beschrieben wurde diese Idee[9] jedoch schon früher. Während diese älteren Ansätze jedoch nur eine Funktion auf einer bestimmten Plattform sicherstellten, ist das *Schily* Makefilesystem Plattform unabhängig.

Die 1994 im *Schily* Makefilesystem eingeführte automatisierte Abstraktion von Plattform und make Programm und die Idee die mit Hilfe von variablen **include** Anweisungen implementierten automatisch Plattformspezifischen Regeleinbindungen stellen jedoch eine bis dahin noch nicht gefundene neue Methode dar.

Un das gewünschte Ziel zu erreichen definiert sich das make Programm zunächst einen Satz von Makros deren Inhalte abhängig von der aktuellen Kompilationsplattform sind. Dies geschieht idealerweise wirklich innerhalb des make Programms. Diese Aufgabe kann allerdings nur dann durch das make Programm erledigt werden, wenn wenn es wie z.B. das Programm **smake** über sogenannte **automake** Eigenschaften verfügt. Ein Teil dieser **automake** Eigenschaften bewirkt, daß das **smake** Programm unter Verwendung der Betriebssystemfunktionen **uname**(2), **sysinfo**(2) und vergleichbaren Schnittstellen Werte in vordefinierten Make Makros setzt. Das Setzen dieser Makros kann allerdings auch durch herkömmliche make Programme erfolgen, wenn sie in der Lage sind das Ergebnis eines Shell Kommandoaufrufes in einem Make Makro zu plazieren. Dabei wird durch Verwendung des POSIX.1 Kommandos **uname**(1) oder durch **arch**(1) und vergleichbare Kommandos eine Systemspezifische Zeichenkette gebildet.

---

[8] Siehe auch die Man Page für **smake** für weitergehende Informationen.

[9] The fourth generation make (Blenn Fowler AT&T Bell Labs 1984)

Aus den so definierten Variablen werden dann im nächsten Schritt plattformabhängige Dateinamen zusammengesetzt, die in Verbindung mit **include** Anweisungen in den Makefiles verwendet werden. Die Includedateien mit make Regeln, die auf diese Weise referenziert werden, beinhalten in plattformspezifischen Dateinamen der jeweiligen Plattform angepasste Regeln. So lassen sich elegant komplexe Regelsysteme definieren.

Nachdem die oben beschriebene Initialisierung der Make Regeln durchgeführt wurde, überprüft das make Programm über nun bekannte Regeln ob zunächst eine Autokonfiguration (z.B. durch Rufen des Shell Scriptes **configure** welches als Ergebnis des **autoconf** Programmes vorhanden ist) durchgeführt werden muß. Nachdem gegebenenfalls die automatische Autokonfiguration durchgeführt wurde, liegen alle benötigten Regeln vor um eine Plattformspezifische Kompilation durchführen zu können.

Damit nun eine einfache Nutzung des *Schily* Makefilesystems möglich ist, verwendet es eine tabellarische Form um die offensichtlichen Abhängigkeiten der zu erzeugenden Programme zu formulieren. Ein Nutzer des *Schily* Makefilesystems muß lediglich ein zum Appliklationstyp passendes Muster-Makefile aus der Liste der bereitgestellten Musterdateien heraussuchen und mit einer Liste der zum Projekt gehörenden Quelldateinamen versehen.

Wie jedes System das auf eine Vereinfachung der Arbeitsabläufe hinausgeht bringt auch dieses System gewisse Einschränkungen mit sich. Die schwerwiegendste Einschränkung dürfte sein, daß durch jedes Makefile in Quelldateibaum nur jeweils ein Zielprogramm erzeugt werden kann. Diese Einschränkung wird jedoch durch die Vereinfachungen in der allgemeinen Nutzung wieder aufgehoben werden. Ein Zwang gewisse Regeln einzuhalten hat zudem in den meisten Fällen den Vorteil, daß durch die auferlegten Regeln die Programme leichter zu warten sind.

6. An was für Regeln muß ich mich halten

Die Regeln zum portablen Programmieren mit dem *Schily* Makefilesystem lassen sich in drei Kategorien einteilen:

- Includedateien
- Typnamen
- Bibliotheken

Diese Kategorien und die damit verbundenen Aspekte bei der Erlangung von Portabilität werden im Folgenden ausführlich erläutert.

6.1. Includefiles

Es gibt zwar für Include Dateien eine Reihe von auf einander aufbauenden Standards,[10] aber nicht alle POSIX-artigen Betriebssysteme halten sich ausreichend genau daran. Es gibt sogar Betriebssysteme die Definitionen enthalten die den POSIX Anforderungen widersprechen. Dazu gehören leider auch Linux Distributionen aus dem Jahre 2000 oder früher.

Um die sich daraus ergebenden Probleme lösen zu können, ist es nötig schon auf der Ebene der im Programm verwendeten Include Dateien eine Abstraktion einzuführen. Diese Abstraktion sollte, im Gegensatz zu den von der FSF in der offiziellen Autoconf Dokumentation beschriebenen Weise, in einer wiederverwendbaren Form geschehen. Wiederverwendbar ist die Form dann, wenn eine Abstraktion, die für ein bestimmtes Portabilitätsproblem geschaffen wurde nur einmal im System existieren muß und beliebiger Programmcode sie nutzen kann ohne eine Kopie anlegen zu müssen. Dies kann z.B. dadurch geschehen, daß der dazu notwendige Code in einer Bibliothek oder in passenden Icludedateien ist.

Bei dem Entwurf der von den Include Dateien zur Verfügung gestellten Schnittstellen sollte wenn möglich die offizielle POSIX Schnittstelle als Portabilitätsabstraktion angestrebt werden. Ist das nicht erreichbar[11], wird eine Schnittstelle gewählt, die möglichst nahe an der POSIX Definition ist.

Das *Schily*-Makfilesystem stellt für die Abstraktion von Portabilitätsproblemen folgende Include Dateien zur Verfügung:

align.h Diese Datei enthält Definitionen über das bei dem aktuellen Prozessor nötige Alignment. Sie wird während der Autokonfigurationsphase mit Hilfe eines Programms generiert und befindet sich unter dem Pfad *.../incs/<plattform-spezifische-subdir>/align.h* unter dem logischen Root Knoten des *Schily* Makefilesystems. Für die Datei *align.h* gibt es weder in POSIX noch auf den üblichen Plattformen ein äquivalent.

btorder.h Diese Datei enthält Definitionen über die Bit und Byte Reihenfolge auf dem verwendeten System. Für die Datei *btorder.h* gibt es weder in POSIX noch auf den üblichen Plattformen ein äquivalent.

---

[10] POSIX.1-1990, UNIX-98 sowie POSIX.1-2001

[11] z.B. wenn eine Plattform als Portierungsziel interessant ist, auf dieser Plattform aber Definitionen vorgefunden werden, die dem POSIX- Standard widersprechen.

ccomdefs.h Dies ist eine vom *Schily* Makefilesystem intern verwendete Datei die solche Definitionen entält, die abhängig vom C-Kompiler sind. Zur Zeit sind dort Definitionen zur Aktivierung der Überprüfung der printf() Formate, in Implementationsvarianten für den GNU C-Kompiler und den SunPRO C-Kompiler, enthalten.

deflts.h Diese Datei wird benötigt, wenn die Funktionen aus der Bibliothek *libdeflt* verwendet werden sollen. Die Funktionen in der Bibliothek *libdeflt* sind entsprechenden Funktionen die unter SYSVr4 vorhanden sind, aber nicht auf allen Plattformen gefunden werden, nachempfunden. Sie dienen dem Parsieren der Konfigurationsdaten aus der Datei **/etc/default/**<*applikations-name*>**.**

device.h Diese Datei wird benötigt, wenn die Makros *major()*, *minor()* oder *makedev()* verwendet werden sollen. Für die Definitionen *major()*, *minor()* oder *makedev()* gibt es zur Zeit keinen geltenden Standard der festlegt welche Includedatei zu verwenden ist.

dirdefs.h Hier findet man Definitionen die benötigt werden wenn die Funktionen *opendir*, *readdir()* und *closedir()* verwendet werden. Nach aktuellem Standard entspricht diese Datei dem Inhalt von *dirent.h*. Ältere Plattformen verwenden jedoch teilweise andere Dateien wie z.B. *sys/dir.h*. Zu beachten ist, daß portable Programme auf keinen Fall das Strukturelement *dirent->d_namlen* verwenden dürfen, da dies nicht Bestandteil von POSIX ist. Stattdessen sollte das Makro *DIR_NAMELEN(dirent)* aus *dirdefs.h* verwendet werden.

fctldefs.h Hier findet man Definitionen für Parameter der Funktionen *open()*, *creat()* und *chmod()* die ja nach Plattform in *fcntl.h* oder in *sys/file.h* zu finden sind.

getargs.h Diese Datei wird benötigt, wenn die Funktionen *getargs()*, *getallargs()* oder *getfiles()*. aus der *schily* Bibliothek *libschily* zum Parsieren von Optionen und Dateinamen der Kommandozeile und verwendet werden sollen. Es handelt sich hierbei um Funktionen die älter als die vergleichbare POSIX Funktion *getopt()* sind und die in ihrer Funktionalität der POSIX Funktion *getopt()* weit überlegen sind.

getcwd.h stellt eine Abstrakte Schnittstelle zur Ermittlung des aktuellen Arbeitsverzeichnisses zur Verfügung. Der Programmcode enthält dabei immer einen Aufruf der Funktion *getcwd()* unabhängig von der aktuellen Plattform.

intcvt.h In dieser Includedatei befinden sich Definitionen die für den Zugriff auf integer Variablen benötigt werden die sich in *Network Byte Order* befinden und deren Startadresse möglicherweise nicht korrekt für die aktuelle Plattform ausgerichtet sind.

jmpdefs.h Häufig stehen Programme vor der Aufgabe das *longjmp()* Sprungziel einer Ausnahmebehandlung in Abhängigkeit vom aktuellen Kontext dynamisch umsetzen zu können. Dazu werden üblicherweise globale Variablen vom Typ *jmp_buf* verwendet. Da der Typ *jmp_buf* in vielen Fällen ein Array ist, kann jedoch in C eine Zuweisung des Variableninhalts in diesem Fall nicht durchgeführt werden. Die Datei *jmpdefs.h* stellt einen Typ *jmps_t* zur Verfügung der eine Struktur mit einliegendem jmp_buf darstellt und daher unabhängig vom aktuellen Typ von *jmp_buf* durch eine C-Zuweisung kopierbar ist.

libport.h Diverse Funktionen (wie z.B. *gethostid()*, *gethostname()*, *getdomainname()* und *usleep()* sind nicht auf allen Betriebssystemen verfügbar. Sie sind jedoch für den Fall daß eine bestimmte Plattform eine bestimmte Funktion nicht enthält in der *libschily* implementiert. Quelldateien, die diese Funktionen aus der *libschily* nutzen wollen sollten die Datei *libport.h* includieren um Zugriff auf die nötigen Prototypen zu erhalten.

maxpath.h Diese Datei stellt statische, plattformunabhängige Versionen der Makros *MAXPATHNAME* und *MAXFILENAME* zur Verfügung. Solange ein Programm auf pre-POSIX Systeme portiert werden soll und nicht von den dynamischen POSIX Werten, die mit *pathconf()* und *fpathconf()* zu erhalten sind, abhängt sind die Makros aus *maxpath.h* ausreichend.

mconfig.h Dies ist die zentrale Datei zur Steuerung der dynamischen Autokonfiguration. Sie sollte von jeder portablen C-Quelldatei includiert werden und sie sollte die erste Datei sein, die in dieser Quelldatei includiert wird.

mmapdefs.h abstrahiert von den unterschiedlichen Varianten der *mmap()* Implementierung auf diversen Plattformen. Wenn das Programm nicht nach Apollo Domain/OS portiert werden soll, dann ist es nicht nötig den zweiten Parameter durch das Makro *mmap_sizeparm(size)* zu übergeben. Das folgende Code Beispiel zeigt wie man in einer portablen Form shared memory alloziert. Diese Methode ist auf den Betriebssystemen wo sie funktioniert (das sind jene auf denen *mconfig.h* das Makro *HAVE_SMMAP* definiert der Methode unter Verwendung SYSVr4 *shmget()/shmat()* vorzuziehen.

```
#include <mconfig.h>
#include <mmapdefs.h>
char *
mkshare(size)
        int     size;
{
        int     f;
        char    *addr;

#ifdef  MAP_ANONYMOUS
        /*
         * No open file /dev/zero needed or allowed (depending on OS).
         */
        f = -1;
        addr = mmap(0, mmap_sizeparm(size),
                     PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, f, 0);
#else
        if ((f = open("/dev/zero", O_RDWR)) < 0)
                comerr("Cannot open '/dev/zero'.\n");
        addr = mmap(0, mmap_sizeparm(size),
                         PROT_READ|PROT_WRITE, MAP_SHARED, f, 0);
#endif
        if (addr == (char *)-1)
                comerr("Cannot get mmap for %d Bytes on /dev/zero.\n",
                                                                size);
        close(f);

        if (debug) errmsgno(EX_BAD,
                        "shared memory segment attached at: %p size %d\n",
                                                (void *)addr, size);

        return (addr);
}
```

patmatch.h

Diese Datei enthält Prototypen und Definitionen für einen kleinen aber sehr effektiven Patternmatchers. Die Funktionen dieses Patternmatchers heißen *patcompile()*, *patmatch()* und *patlmatch()*. Sie können genutzt werden wenn die Applikation gegen die *libschily* linkt, denn der Patternmatcher ist Bestandteil der *libschily*.

prototyp.h

Diese Datei ist einer interner Bestandteil des *Schily* Makefilesystems und stellt Makros zur Verfügung die es möglich machen daß ein K&R C-Kompiler Dateien kompilieren kann, obwohl sie Funktionsprototypen enthalten. Die Datei *prototyp.h* muß nicht explizit includiert werden, denn die wird durch *mconfig.h* automatisch includiert.

schily.h

Diese Datei stellt Prototypen für die Funktionen aus der Bibliothek *libschily* zur Verfügung.

sigdefs.h

Diese Datei stellt eine Abstraktionsebene zur Verfügung, die es ermöglicht sowohl auf BSD, als auch auf POSIX konformen Systemen Signale zu blockieren.

standard.h

Diese Datei enthält diverse Definitionen die die Programme besser lesbar machen (wie z.B. *LOCAL*, *GLOBAL*, *EXPORT*, sowie *TRUE* und *FALSE*). Außerdem enthält sie Definitionen, die es ermöglichen eine Kompilation auch mit einem *K&R* -Kompiler durchzuführen. Die Datei *standard.h* sollte nach den Dateien *stdxlib.h* und *unixstd.h* includiert werden.

statdefs.h

Da auf einigen Betriebssystemen in *sys/stat.h* nicht alle der von *POSIX.1* geforderten Makros definiert sind und teilweise sogar fehlerhafte Definitionen anzufinden sind, ist es sinnvoll an Stelle von *sys/types.h* und *sys/stat.h* besser *statdefs.h* zu versenden. So ist sichergestellt, daß alle erwarteten Makros in funktionstauglicher zur Verfügung stehen.

stdxlib.h

Auf sehr alten Betriebssystemen ist die Datei *stdlib.h* noch nicht vorhanden. Daher sollte die Datei *stdxlib.h* verwendet werden, die je nachdem ob sie auf der aktuellen Plattform vorhanden ist, entweder *stdlib.h* verwendet oder die wichtigsten Definitionen selbst innerhalb von *stdxlib.h* definiert bereitstellt.

stkframe.h

ist eine Datei die für Programme interessant ist, die Zugriff auf den sogenannten *Framepointer* haben wollen. Da solche Programme nur von erfahrenen Programmierern geschrieben werden können wird hier für weitere Information auf die Lektüre des Inhalts der Includedatei verwiesen.

strdefs.h

BSD artige Betriebssysteme haben *strings.h* während POSIX konforme Systeme *string.h* bereitstellen. Die Verwendung von *strdefs.h* löst dieses Problem und stellt eine durch ihr Vorhandensein im *Schily* Makefilessystem eine universelle Includedatei dar.

termcap.h

Diese Datei stellt Definitionen für die *schily* Bibliothek *libxtermcap* zur Verfügung. Bei der Bibliothek *libxtermcap* handelt es sich um eine vollkommen eigenständige **termcap** Implementierung. Dadurch wird es möglich wirklich portable Applikationen zu entwickeln die von den Fähigkeiten der Terminals abstrahieren müssen. Wird statt der Bibliothek *libxtermcap* die **termcap** Bibliothek verwendet die sich auf dem System befindet, dann gibt es häufig Probleme die dadurch entstehen daß die **termcap** Bibliothek des Systems nur ein Verweis auf die *terminfo* Bibliothek ist, die ihrerseits leider auf einer binären Datenbank basiert. Diese Datenbank ist zwar theoretisch plattformunabhänig, in der Praxis jedoch nicht, denn nur eine zentrale Autorität könnte solch eine Plattformunabhängigkeit garantieren. In Verbindung mit der Tatsache daß häufig bestimmte kommerzielle Systeme bewusst fehlerhafte Einträge für Terminaltypen von Konkurenzfirmen enthalten, ist die Verwendung von **terminfo** problematisch in der Portabilität. Abhilfe schafft die Bibliothek *libxtermcap* die, wie die leider nicht portable **termcap** Implementierung von BSD, einen Suchpfad zu der **termcap** Datenbasis implementiert und so in der Lage ist zuerst in einer privaten Datei *~/.termcap* zu suchen und für den Fall, daß kein Eintrag gefunden wurde, danach in **/etc/termcap** sucht. Da die **termcap** Datenbasis eine Textdatei ist, ist sie im Gegensatz zur **terminfo** Implementierung absolut Portabel.

timedefs.h

Die Verwendung von *time.h* und bezw. oder *sys/time.h* bereitet immer wieder Probleme. Wird stattdessen *timedefs.h* verwendet werden die richtigen Dateien in der richtigen Reihenfolge includiert.

unixstd.h

Auf sehr alten Betriebssystemen ist die Datei *unistd.h* noch nicht vorhanden. Daher sollte die Datei *unixstd.h* verwendet werden, die je nachdem ob sie auf der aktuellen Plattform vorhanden ist entweder *unistd.h* verwendet oder die wichtigsten Definitionen selbst bereitstellt.

utimdefs.h

Programme, die die Zugriffszeiten von Dateien aktiv modifizieren wollen, sollten *utimdefs.h* includieren. Die dort enthaltenen Definitionen werden benötigt, falls die verwendete Plattform nicht die Funktion *utimes()* und stattdessen nur *utime()* zue Verfügung stellt.

utypes.h

Hier werden benutzerdefinierte Typen zur Verfügung gestellt. Neben anderen Typen stellt *utypes.h* im Wesentlichen eine Abstraktion der UNIX-98 bzw. C-99 Definitionen aus *inttypes.h* bereit. Leider gibt es Betriebssysteme, die nicht UNIX-98 konform sind und Typen gleichen Namens wie die offiziellen Typen aus *inttypes.h* jedoch mit unterschiedlicher Bedeutung definieren. Daher lassen sich die offiziellen Typen aus den Standards in portablen Programmen nicht verwenden. Die Lösung bieten abstrakte neue Typnamen in *utypes.h* die im Prinzip so heißen wie im Standard jedoch die ersten Buchstaben in Großschreibung verwenden.

vadefs.h

Ältere Betriebssysteme bzw. Kompiler haben die Datei *varargs.h* während neuere Systeme bzw. Kompiler die Datei *stdarg.h* zur Verfügung stellen. Einige Betriebssysteme haben beide Dateien und es ist oft nicht leicht zu entscheiden, welche der beiden Dateien in diesem Falle sinnvollerweise verwendet werden sollte. Dieses Problem wird durch die Verwendung von *vadefs.h* gelöst. Die Tatsache, daß der im *varargs.h* bzw. *stdarg.h* definierte Typ *va_list* unter Umständen durch ein Array dargestellt werden kann, bereitet weitere Probleme bei der Portierung von Programmen. Diese Probleme entstehen durch die Art und Weise wie POSIX.1-1990 den Inhalt der Datei *stdarg.h* definiert und dabei den Typ, der sich hinter *va_list* verbirgt, offenläßt. Da Arraytypen in der Sprache C sich nicht nach den

Regeln aller anderen Typen verhalten, ist es unmöglich mit Mitteln von POSIX.1-1990 den Inhalt einer Variablen vom Typ *va_list* zu kopieren. Zum Kopieren von Variablen des Typs *va_list* stellt *vadefs.h* das Makro *va_copy()* zur Verfügung welches auch im C-99 Standard definiert ist, jedoch bislang nur unter Solaris-2.5.1 oder neuer auch tatsächlich verfügbar ist. Ein anderes Problem, welches sich gelegentlich stellt, ist bisher durch keinen der bekannten Standards behandelt. Es gibt bisher keine offizielle Möglichkeit um zuverlässig eine Variable vom Typ *va_list* aus einer Vararg Parameterliste einer Funktion zu extrahieren. Dies ist jedoch durch die Verwendung des Makros *__va_arg_list(list)* aus *vadefs.h* möglich.

waitdefs.h

Einige Betriebssysteme stellen *wait.h* andere *sys/wait.h* und andere beide Dateien zur Verfügung. Einige Systeme verwenden einen Integer als Parameter für die *wait()* Funktion und diverse Makros wie z.B. *WEXITSTATUS*, andere eine Struktur bzw. Union. Die Datei *waitdefs.h* löst dieses Problem und stellt einen abstraken Typ *WAIT_T* und alle in *POSIX.1* definierten Makros zur Verfügung.

xmconfig.h

In dieser Datei befinden sich die Ergebnisse eines *autoconf* Laufes. Sie wird während der Autokonfigurationsphase generiert und befindet sich unter dem Pfad *.../incs/<plattform-spezifische-subdir>/xmconfig.h* unter dem logischen Root Knoten des *Schily* Makefilesystems und wird automatisch durch die Datei *mconfig.h* includiert.

6.2. Typnamen

Bis auf wenige Ausnahmen ist es möglich die üblichen Typen zu verwenden die auch in den Standards beschrieben werden.

Zu den Typen, die verwendet werden können und zur Erlangung einer hohen Portabilität auch sollten gehören folgende POSIX Typen die mit einer Ausnahme alle in **sys/types.h** definiert werden.

caddr_t

Für Speicheradressen, wird hauptsächlich bei der Kommunikation mit dem Kern verwendet. Dieser Typ ist nicht durch den POSIX Standard abgedeckt, wird aber in allen bekannten UNIX Implementierungen bereitgestellt und in vielen Schnittstellen verlangt.

daddr_t
Für Festplattenadressen, wird hauptsächlich bei der Kommunikation mit dem Kern verwendet. Dieser Typ ist nicht durch den POSIX Standard abgedeckt, wird aber in allen bekannten UNIX Implementierungen bereitgestellt und in vielen Schnittstellen verlangt.

dev_t
Für Gerätenummern wie sie bei **mknod**() verwendet werden und wie sie in **struct stat** enthalten sind.

gid_t
Für Benutzergruppennummern.

ino_t
Für Dateiidentifikationsnummern wie sie in **struct stat** enthalten sind. Nach POSIX ist das Paar aus **st_dev** und **st_ino** auf einem System eine eindeutige Möglichkeit eine Datei zu beschreiben.

mode_t
Für Dateityp und Zugriffsrechte wie sie in **struct stat** enthalten sind.

nlink_t
Für den Hardlinkzähler wie er in **struct stat** enthalten ist.

uid_t
Für Benutzernummern.

off_t Für Offsets in Dateien und Dateigrößen. Hier ist besonders zu beachten, daß es Probleme geben kann wenn die Kompilation auf einem **Large File** fähigen System erfolgen soll und das zu erzeugende Programm kein 64 Bit Programm ist. In diesem Fall lassen sich Dateioffsets nicht mehr durch einen **long** darstellen.

pid_t
Für Prozessidentifikationsnummern.

size_t
Zur Darstellung der Größe von Objekten im Speicher.

ssize_t
Zur Darstellung der Größe von Objekten im Speicher wenn eine Fehlerbedingung als Rückgabewert möglich ist.

time_t
Für die Zeit in Sekunden.

socklen_t

Dies ist der Einzige Typ, der außerhalb der üblichen Typdefinitionen vorgefunden wird. Er bezieht sich auf Längenangeben wie sie in Kommunikationsstrukturen bei der Netzwerkprogrammierung verwendet werden.

Die Sprache C definiert selbst keine keine Basystypen, die eine vordefinierten festen und Plattformunabhängigen Wertebereich besitzen. Daher ist es unter ausschließlicher Verwendung von C Basistypen nicht möglich Schnittstellen Plattformunabhängig eindeutig zu beschreiben.

Die Standards C-99 und UNIX-98 (SUSv2) sowie POSIX.1-2001 definieren für die Portabilität wichtige Typen mit festgesetztem Wertebereich in der Datei **inttypes.h**. Diese Datei ist allerdings auf vielen aktuellen Systemen noch nicht vorhanden und einige ältere Plattformen enthalten sogar Typdefinitionen, die den Typen im Standard widersprechen. Die Typen die in dieser Datei im Einzelnen definiert werden sind:

int8_t, int16_t, int32_t, int64_t, intmax_t, intptr_t, uint8_t, uint16_t, uint32_t, uint64_t, uintmax_t, uintptr_t.

Da wie oben bereits angeführt Überschneidungen der Typnamen mit nicht Standardkonformen Systemen vorliegen, ist es zu empfehlen Anstelle von **inttypes.h** die Datei **utypes.h** aus dem *Schily* Makefilesystem zu verwenden. Dort werden vergleichbare Typen definiert, jedoch mit leicht abweichenden Namen so daß die oben erwähnten Überschneidungen vermieden werden. Die in **utypes.h** definierten Typen lauten:

Int8_t, Int16_t, Int32_t, Int64_t, Intmax_t, Intptr_t, UInt8_t, UInt16_t, UInt32_t, UInt64_t, UIntmax_t, UIntptr_t.

In der Datei **utypes.h** wird dabei - falls vorhanden - die Datei **inttypes.h** verwendet. Ist die offizielle Datei **inttypes.h** auf der aktuellen Plattform nicht verfügbar, dann werden benutzerdefinierte Typen zur Verfügung gestellt, die mit Hilfe von GNU autoconf Testergebnissen spezifiziert werden.

6.3. Bibliotheken und Bibliotheksroutinen

Einige Funktionen dir man gerne in portablen Programmen verwendet sind nicht bei allen Plattformen verfügbar. Zu diesen Funktionen gehören z.B. **gethostid**(), **gethostname**(), **getdomainname**(), **getpagesize**()und **usleep**(). Will man hier portabel programmieren hat man die Möglichkeit entweder an allen Aufrufstellen seines Programms bedingten Code zu verwenden oder eine Abstraktion dadurch einzuführen diese Funktionen im Falle der Nichtverfügbarkeit zu emulieren.

Die Methode fehlende Funktionen auf bestimmten Plattformen zu emulieren kann Probleme bereiten falls es zu stark unterschiedliche Prototypen auf den Plattformen gibt, die die betreffende Funktion unterstützen. Für den Fall, daß eine weitgehend einheitliche Schnittstelle verwendet wird hat die Methode der Emulation fehlender Funktionen jedoch den großen Vorteil, daß der Quellcode des Programms besser lesbar bleibt, denn die oft verwirrenden Emulationen befinden sich innerhalb separater Dateien. Die Emulationen sollten sinnvollerweise in einer Bibliothek gesammelt werden.

7. Ein kleines Kochbuch für Einsteiger

Um der Theorie in den vorangegangenen Kapiteln nicht überhand nehmen zu lassen, erscheint es sinnvoll ein paar praktische Ratschläge für den Umgang mit den Portabilitätshilfsmitteln des *Schily* Makefilesystems zu geben.

7.1. Erste Schritte mit dem Schily Makefilesystem

Der erste Schritt zur Verwendung besteht immer darin, eine aktuelle Version des *Schily* Makefilesystems zu installieren. Die Distribution des *Schily* Makefilesystems besteht aus einem TAR Archiv. Nach dem Auspacken dieser Distribution findet man einen kompletten Dateibaum mit diversen Dokumentationen sowie ein weiteres TAR Archiv mit dem Namen **makefiles.tar.gz** welches einen Skelettdateibaum für eine neues Prokjekt enthält.

Wird dieses TAR Archiv ausgepackt, dann entsteht ein Skelettdateibaum in den nur noch an passender Stelle ein Verzeichnis für die neu zu erstellenden Quelldateien angelegt werden muß. Nachdem dann noch aus dem Verzeichnis **TEMPLATES** ein zum Typ der neuen Applikation passendes Makefile herausgesucht und im neuen Quellverzeichnis installiert ist kann **make** eingegeben werden um eine Kompilation durchzuführen.

Dabei wird das *Schily* Makfilesystem zunächst feststellen daß noch keine Autokonfiguration durchgeführt wurde und daher ein durch GNU autoconf generiertes **configure** Shell Script starten. Nachdem danach eventuell notwendige weitere Autokonfigurationsschritte durch das **make** Programm durchgeführt wurden beginnt die Kompilation der eigenen Quelldateien.

7.2. Progammierrichtlinien zur Verwendung mit dem Schily Makefilesystem

Ein unter Verwendung des *Schily* Makefilesystems geschriebenes C-Programm includiert generell zuerst die Datei *mconfig.h.* includieren, danach kommen die restlichen Includefiles. Wichtig ist dabei, daß wie schon beschrieben nicht bevorzugt die offoziellen Systeminclude Dateien verwendet werden sondern - falls vorhanden - themenspezifische Includedateien aus dem Schily Makefilesystem.

```
#include <mconfig.h>
#include <stdio.h>
#include <stdxlib.h>
#include <unixstd.h>
```

Inhaltsverzeichnis