Development Tutorial (a.k.a Build FAQ)

by Marco van de Voort

March 24st 2010

Vorwort zum Development Tutorial

FAQ version 0.08

FPC version 2.4.0/2.5.1 (release/devel)

Diese FAQ wurde in den frühen 2.0 Zeiten (2005) erzeugt, als ich merkte, dass immer mehr Menschen, die mit den weiter fortgeschrittenen Aspekte des buildprocesses des FPC, wie die Beobachtung des täglichen SVN Status, Crosscompiling beschäftigten. Sie versuchten generell Dinge zu tun, die nur von einen kleinen Kreis der Kern-Entwickler, vielleicht einige wenige auf den Mailinglisten und in dem Lazarus-Projekt gemacht werden. Dinge wie Bootstrapping, eine neue Version, Release Building, Crosscompiling, Parametrierung der Build-Prozess, Anschluss an neue Betriebssysteme und vieles mehr werden versucht. Dies ist logisch, denn durch (hauptsächlich) Peter's bemühungen ist der Building Prozess sehr viel robuster und flexibler geworden. und kann verwendet werden. Seit der ersten Version dieser FAQ, habe ich speziell den crosscompiling Teil des Prozesses ein bisschen mehr korrigiert.

Das development Tutorial ist als Nachfolger für die alte 0.99x make cycle FAQ gedacht, welche fürchterlich veraltet war. Nochmehr als die Basisidee der make cycle FAQ nicht in der Praxis funktionierte. Diese FAQ beinhaltete nur eine Schritt für Schritt Anleitung der Basisfeatures und wenn etwas schief ging (z.B. eine .ppu Datei einer alten Version irgendwo), so hatten die Benutzer oft keine Ahnung wie sie das Problem lösen sollten. So wurde diese FAQ erstellt um einen tieferen Einblick in das erstellen des FPC und verwandten Themen zu geben. Mit viel mehr an Hintergrundinformationen welche beim generellen Problemlösungsprozess hilft. Wenn Sie jetzt denken das diese FAQ ist zu detailiert und zu pedantisch und sollte auf die nackten Grundlagen reduziert sein, so ist meine Antwort: War vorhanden, ist getan worden, hat nicht funktioniert :-)

Diese Informationen sind kein Ersatz für die richtigen Anleitungen. Viele von den Informationen hier befinden sich auch in den normalen Anleitungen, manche Information sind ein Ausblick auf später Versionen. Versuchen sie die Anleitung komplett durchzulesen, auch diese Teile welche derzeit nicht für sie wichtig sind. Es ist alles so zusammengesetzt, das auch die derzeit nicht so wichtigen Teile helfen, das sie ein besseres Überbicksverständnis bekommen.

Der Zweck dieser FAQ ist verschieden von den Anleitungen, weil dieses Dokument eher als Benutzerhilfe als ein Nachschlagewerk gedacht ist und nicht vollkommen synchron mit der Compilerversion ist.

Wenn sie mehr Fragen oder Anregungen haben, verwenden sie die FPC Mailliste http://www.freepascal.org/maillist.html oder den IRC-Kanal [1]

Versionierung der FAQ

Die FAQ kommt in zwei Versionen, als PDF-Dokument unter http://www.stack.nl/marcov/buildfaq.pdf udn als HTML-Version unter http://www.stack.nl/marcov/buildfaq. Die PDF-Version ist die gültige und wird öfters erneuert. Die HTML Version wird mehr verwendet um ULR zu speziellen Themen auf der Mailingliste und im IRC zu posten. Zusätzlich ist der HTML-Export von LyX ist nicht der stabilste, so kommt es zeitweise zu Fehlversuchen.

- Versions 0.01 und 0.02 werden laufend auf Stand gebracht, beide existieren in den verschiedensten Versionen.
- Version 0.03 ist hauptsächlich ein Update für das $FPCTARGET und den entsprechenden Verzeichnislayoutänderungen in der Version 1.9.5, und erweitern den Index und Glossar erheblich. Zusätzlich sind die 1.0.x Topics ausgelaufen.
- Version 0.04 ist ein Update für 2.0 und später als 2.0 Entwicklung. Peter hat große Pläne mit dem Build Prozess (Ersetzen von MAKE mit einer mehr FPC freundlichen Lösung), so 0.05 kann ein großer Sprung werden. Genauso werden SVN Anleitungen benötigt.
- Version 0.05 ist ein Update nach Prozessänderungen beim Crosscompilieren und dem einführen eines internen Linker
- Version 0.06 ist ein Update nach einer großen Pause nach dem Verluste der Lyx Quellen durch einen Computerdiebstahl. Eine PDF Version wurde mittels Schrifterkennung eingelesen, neu formatiert und einige Neuerungen wurden gemacht:
  - Compiler Versionsnummern auf Stand 2.2.2 gebracht
  - Neue Paket Struktur.
  - Index erweitertis a maintenance release after
  - Mehr 1.0.10 und CVS entfernt
- Version 0.07 ist eine Wartungsausgabe nach der 2.2.4 Freigabe

[1] irc.freenode.net channel #fpc, am meisten besucht in der späten CET-Zeit.

  - Compiler Version auf Stand gebracht
  - einige 2.3.x Themen, welche in der Zukunft erweitert werden.
  - Lyx 1.6.2 (mehr Sortierung der all-caps Index Einträge)
  - Neuere Buildscripts

- Updates 0.07a und b habe nur kleiner Schreibfehler ausgebessert und wurden in den Tagen nach der Initialen 0.07 Version herausgebracht
- Version 0.08 ist eine Verbesserungsversion nach dem herauskommen der 2.4.0 Version und LyX Version 1.6.5

Contents

Part I

Normales Erzeugen (Ordinary building)

Chapter 1

Grundlagen, Namensgebung, Versionen, Voraussetzungen usw.

1.1 Versionen, Zweige, Bennenung

Grundlegend gibt es fünf Versionsklategorien des FPC jetzt:

pre-1.0 Diese sind Versionen welche als 0.99.x Versionen bezeichnet werden. Versionen vor 0.99.8 haben keine Delphi Erweiterungen. Generell werden diese nicht mehr unterstützt, ausser speziellen Versionen (0.99.12 und 0.99.14 basierende Versionen), welche wichtige Beta's für den 1.0 waren. Wie auch immer, wir sprechen hier von über 10 Jahren alte Betas hier, und jede Menge hat sich geändert durch die 1.0x Serie.

1.0.x Der sogennante FIXES_1_0_0 Zweig. Diese Versionen (1.0, 1.0.2, 1.0.4, 1.0.6 und 1.0.10) sind Bugfix Releases des Basis 1.0. Wie auch immer, TP und Delphi Kompatibilitätsverbesserungen können die Rückwärtskompatibilität brechen. 1.0.x ist eingefroren und es gibt keine neuen Versionen oder Bugfixes nach dem 1.0.10 vom Mai 2003 und wir empfehlen dringend das Upgrade auf die 2.0

1.1.x/1.9.x/2.0.x Da sist der Zweig zur 2.0.x Serie. Er wird als Development Zweig vor der 2.0,0 bezeichnet, allerdings ist jetzt der 2.0 als stabil erklärt, der beste Name ist also 2.0.x stable branch. Verglichen mit dem 1.0.x Zweig, beihaltet er die meisten der fehlenden Delphi Features (dynamic arrays, interfaces, default parameters) und unterstützt etliche Prozessoren. Dieser Zweig hat verschiedene Versionsnummern, während er in Alpha wurde er 1.1.x benannt, während der Beta war er 1.9.x und als reales Release wurde er 2.0.x gennant. Dieser Zweig ist jetzt geschlossen, 2.0.4 ist geplant die letzte Veröffentlichung des FIXES_2_0_0 Zweig

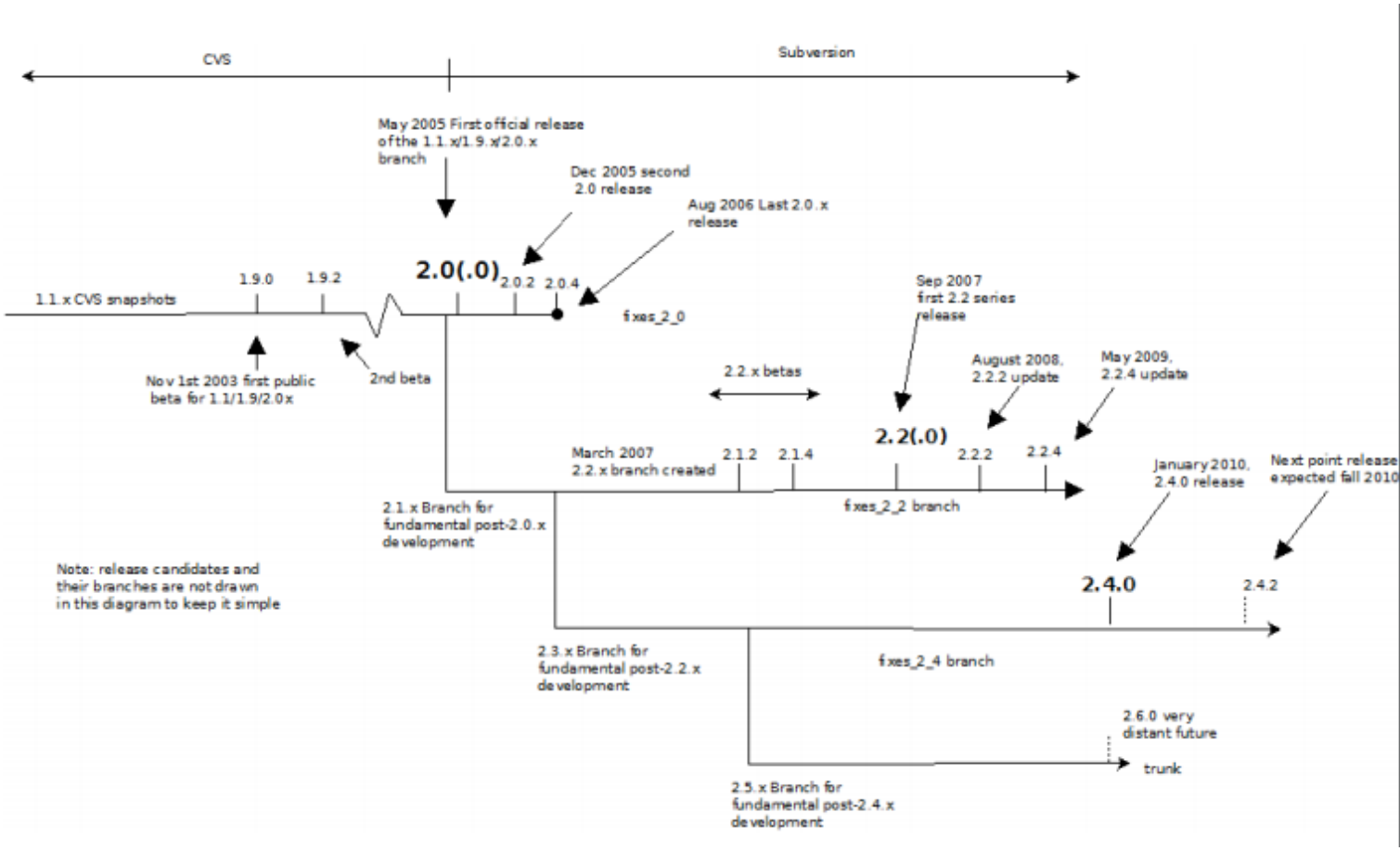
2.1.x/2.2.x Dies ist der Zweig für die FPC 2.2.x Serie. Die Hauptverbesserung sind die Unterstützung für das interen Linken unter Windows (win32/win64/wince). Win64 und WinCE sind auch neu. fixes_2_2 Zweig. 2.2.4 ist der letzte des 2.2.x Zweiges, weil die Zusammenführung der verschiedenen Zweige immer unmöglicher wurde.

2.3.x/2.4.x Der Zweig der nachfolgenden 2.4 Serie. 2.4.0 wurde veröffentlich am 1 Jänner, 2010 und beinhaltete Verbesserungen in Resourcen Handling, dwarf und für neuere Architekturen (speziell die 64-bit Architektur)

2.5.1 Das ist die version in TRUNK, der Hauptzweig für die Entwicklung. Einige der Erweiterung werden in den 2.4.x Zweig zurückgeführt, während andere wiederum den Endbenutzer erst erreichen werden, wenn der 2.5.1 fertig wird als 2.6.0 oder 3.0.0.

Man kann sich ansehen welche (nach 2.0.0) Zweige existierung, mittels des `http://svn.freepascal.org/cgi-bin/viewvc.cgi/?root=fpc` Viewvc Webinterface.

Veröffentlichungen (Releases) werden makiert mit RELEASE_x_y_z für die Version x.y.z. Somit ist Version 1.0.2 als RELEASE_1_0_2 bezeichnet im CVS. Die Version 1.0.8 möge im CVS vorhanden sein, allerdings gibt es keine offizielle Veröffentlichun von 1.0.8. Dies ist bedingt dadurch das die Erzeugung der 1.0.8 Version eine lange Zeit in anspruch nahm und die Pakete viele male neu erzeugt wurden, während sie über FTP herunterladbar waren. NCurrently it looks like the first release of theachdem das Coreteam befürchteten , das das Paket mit den vielen Versionen nur Verwirrung stiften kann, wurde entschlossen die Versionsnummer auf 1.0.10 zu erhöhen. Die Versionsgeschichte ist verdeutlicht im folgenden Graphen:



Währen 1.0.x nur zwei Prozessortypen (intel x86 und Motorola m68k) unterstützt, beinhaltet der 1.1.x Zweig mehr grundlegend Ansatz um den FPC zu einem mehrfach architektur unterstützenden Compiler zu machen. Es schaut derzeit so aus als würde die erste Veröffentlichung des 1.1.x Zweigs, x86, ppc, Arm (Zaurus und gleRequirementsichwertige Geräte), AMD64(x86-64) und Sparc (V8,V9) unterstützen wird.

1.2 Voraussetzungen

Ein unterstütztes Betriebssystem und ein unterstützter Prozessor sind die Hauptanforderunegn. Diese Informationen werden besser von der Webseite direkt nachgelesen, weil der Status der Informationen sich immer wieder ändert.[1]

Der FPC Build Prozeß benötigt verschiede Werkzeuge. Welche normalerweise vom Betriebssystem , oder vom FPC Release Paket bereitgestellt werden. Ich bennene sie hier explizit, um es für Leute mit Build Problemen einfacher zu machen.

Hier die Hauptwerkzeuge:

ld Der GNU Linker. Verbindet (linkt) eine Menge von .o und .a Dateien zusammen, um sie entweder in einer Bibliothek (=library) (.dll or .so) oder einem fertigen Programm (wie .exe unter Windows, ohne Dateierweiterung unter Unix) zu binden.

as Der GNU Assembler. Übersetzt (Assembles) die Textform in einer .s oder .as Datei zu einer .o Datei.

ar benötigt um statische Bibliotheken (.a) aus .o's zu erzeugen. Statische Biblitheken werden benutzt für das Erzeugen beim Smartlinking.

make GNU make. Bestimmt die Reihenfolge in welcher gearbeitet (gebaut = build). Sowohl auf datei als auch auf Verzeichnisebene. Unter *BSD wirde es auch gmake genannt.

ppc386 oder ppc<processor> im Allgemeinen die letzte Veröffentlichte Version. Bootstrapping (Neuerstellung aus dem Nichts) ist mit anderen Kompilern nicht wirklich realistisch und, so mein Wissen, wurde in den letzen Jahren nicht mehr versucht (1.0.x Zeiten)

[1] Aktuell, während ich diese Zeilen hier schreibe, erzeuge ich das erste funktionierende "Hello world" Programm auf einer Sparc V8 Architektur :-)

Die ersten drei findet man im Paket "binutils", die Version ist nicht so wichtig, solange sie nicht vorsintflutlich ist. Zumindest auf den Hauptplattformen. Einige Plattformen paketieren alte Versionen, zum Beispiel OpenBSD verwenden alte Versionen, weil es a.out als binäres Format nutzt. Ich habe gehört das das in Version 3.4 beseitigt sein soll. Dieses Programme sind in seltenen Fällen die Quelle der Probleme, allerdings ist es abhängig vom verwendeten Betriebssystem und Prozessor, welche die Sache verkomplizieren können.

Make wird von denselben Quellen wie die binutils verwendet, aber seperat gepackt. Siehe auch in der seperate section weiter unten für weitere allgemeine Warnungen.

Windows Benutzer welche das System von Grund auf neu erzeugen, sollten die Dateien makew32.zip und asldw32.zip (oder ähnlich) aus der letzen Veröffentlichung (Release) Verzeichnis nehmen. Diese Dateien sind in einem seperaten Verzeichnis zu finden und sind die benötigten externen Werkzeuge.

Bei *Mac OS X/Darwin* , sind die binutils und make ein Teil der Apple Developertools, welche für 10.3 automatisch installiert werden, wenn man XCode installiert. Fink Fink-Projekt ermöglicht die Verwendung von Open-Source-Programmen für die UNIX-basierenden Betriebssysteme auf Computern von Apple mit dem Betriebssystem Mac OS X.) ist nicht unbedingt benötigt um mit FPC zu arbeiten, allerdings verwenden die meisten Unix Bibliotheken, die sie benötigen (wie mysql,ncurses), Teile von Fink. is not strictly required for FPC operation, but most Unix libraries you might need (like mysql, ncurses etc) are part of Fink. Es geht das Gerücht um, das in dedr Version 10.4 von Apples Betriebssystems die Werkzeuge auf der Basis der GNU versionen arbeiten.

1.2.1 ld: Der GNU Linker.

Der GNU Linker ist der letzte Schritt im Erstellungsprozeß von den FPC Quellen hin zum ausführbaren Programm. So wie schon früher gesschrieben, einen aktuelle version welche Linkerscript Dateien unterstützt ist die einzige Voraussetzung für ein einfaches arbeiten, ausser für Win32, wo der Linker -base-file und -image-base Parameter verstehen muß (diese werden seit über 2 Jahren unterstützt). Die unterstüzte Win32 Plattform für die die Buildtolls unterstütz werden, ist mingw32, nicht cygwin. FPC kann auch gegen cygewin Bibliotheken linken.

Man kann auch andere Linker als den GNU verwenden, das erfordert allerdings das alle Teile die den Linker aufrufen, reimplementiert werden müssen (das wurde zum Beispiel für OpenBSD a.out im 1.0.x Compiler gemacht und bei Darwin's mach-O in 2.0+) . Wie auch immer, behalten wir im Kopf das die Verbindung zwischen dem Assembler und dem Linker ein gemeinsames Format für die Objektdatei (.o) ist. Wechseln des Linker zu einem der ein anderes Format verwendet, wird also auch einen anderen Assembler benötigen und umgekehrt, ein anderer Assembler kann Änderungen an Teilen des FPC welcher Assembler Code erzeugt nötig machen. Diese Dinge sind wie immer machbar, auch für Personen die nicht perfekte Compiler Gurus sind.

Wenn die Plattform nicht ein Mainstream *nix ist oder für Windows. Versuchen sie einen Linker zu finden der die –ld-sections Parameter unterstützt. Das neue Smartlinking basiert auf diesen LD Parameter.

Beginnend mit dem FPC 2.1.1 hat der Compiler selbst einen internen Linker für einige Plattformen, welche mittels -Xi eingeschalten werden können. Dieser interen Linker linkt schneller und benutzt weniger Speicher, speziell wen Smartlinking[2] benutzt wird. Während dies hier geschrieben wird, untertsützt die Windows Plattform (PE) ebenso den internen Linker.

1.2.2 as: GNU Asssembler

Der Assembler muß der GNU (G)AS sein, und relativ neu. Wirklich alte x86 GNU Assembler können versteckte Fehler haben, welche nicht sichtbar sind wenn der gcc verwendet wird. Weil GNU AS ist ein typischer Backend Assembler und in den past adressing modes und opcodes welche nicht vom gcc verwendet werden können Probleme bestehen.

Ein Beispiel für das sind die OpenBSD 3.x Serien, wo beim verwenden eines neueren Assembler von den Portzweigen der FPC erzeugt werden kann, während der normale Assembler aus den Bibliotheken fehlschlägt. Eine relativ neue Version ist also sehr angenehm, auch weil sie die neueren x86 Befehle (SSE2,SSE3) kann.

Auf eingen Plattformen (win32 und x86 ELF) hat der FPC einen internen Assembler und vermeidet AS aus Geschwindigkeitsgründen. Dieser interen Assembler wird im FPC Jargon auch binwriter genannt. Der binwriter ist ein wenig mehr als ein reiner Assembler, weil er auch direkt statische Bibliotheken (*.a) schreiben kann. Es kann schon bei einfachen Kompilierungen bemerkt werden, besonders beim Smartlinken spürt man die Geschwindigkeitsvorteile die der binwriter bringt.

FPC hat zusätzlich die Möglichkeit Kode für TASM, NASM, MASM und WASM (watcom) Format zu erzeugen. Diese sind nicht so stark getestet, so kann die Leistung abweichen.

[2] In etwa 250-275 MB als maximale Speichergröße wird für das ganze Smartlinken von Lazarus benötigt. im Gegensatz zu +/- 1.5GB beim GNU LD

1.2.3 ar: GNU Archiver

Der Archiver erzeugt Archivdateien (.a) aus den Objektcodedateien (.o). Das wird hauptsächlich gemacht um die Anzahl der Dateien die im Linkprozeß verwendet werden und auf der festplatte zu reduzieren. Archivdateien werden oft als statische Bibliotheken bezeichnet, weil sie grob betrachtet den selben statischen Kode beinhalten, so wie die .so (oder .dll) Dateien es für das dynamische Linken verwenden (eine .dll und .so kann mehr als eine .o beinhalten). Der GNU Linker kann direkt auf die .a Bibliotheken zugreifen.

AR kann zeitweise ein Problem sein, weil der Compiler übergibt alle einzelnen Dateinamen auf der Kommandozeile. Während des so genannten Smartlinkens können das wahrhaft viele sein, mehr als die auf dem entsprechenden Betriebssystem erlaubte Anzahl von Parameter. (64k Parameter sind zu wenig, 128k jetzt in Ordnung) als Voraussetzung.

1.2.4 make: GNU make

Der Build Prozeß von Free Pascal benutzt reines (GNU make) makefiles, welche vom FPCmake erzeugt werden. Das Makefile.fpc ist eine einfache INI Datei welche die globale Vorlage (Template) [3] erweitert und mit Parametern versorgt. Es gibt Pläne für die Zukunft sich vom MAKE utility zu befreien udn alles zusammen in einer eignenen, mehr spezialisierten Version zumachen, allerdings ist das noch im Anfangszustand. Das jetzige System ist auch schön und flexibel.

Das jetzt verwendete make ist GNU make, es sit für nahezu alle Plattformen vorhanden. Wie auch immer, im folgenden für einige Plattform spezifische Spezialitäten.

Linux Das einzige mit wenigen Spezialitäten. Bei allen Unixes, Linux verwendet eine Menge an GNU Werkzeugen und viele dieser Werkzeuge sind von den originalen Unix abgeleitet. Wenn dort ein Make oder Make-Paket ist, so ist es normalerweise von GNU.

***BSD** Das standard make ist die pmake Variante, welches eine leicht abweichende von GNU ist. Die makefile Vorlage die derzeit vom FPC verwendet wird, ist derzeit nicht kompatibel mit pmake. (Hinweis, Hinweis) GNU make ist von dem ports Zweig, normalerweise als devel/gmake. Vergessen sie nicht, es ins selbe bin Verzeichnis des ports-$PREFIX (/usr/local/bin, /usr/pkg/bin etc) in ihrem Pfad zu geben. Zusätzlich müssen sie alle make durch gmake ersetzen. In der Proaxis ist das kein großes Problem, weil gmake ist gewöhnlich Verfügbar auf BSD Systemen, installiert als eine Abhängigkeit von vielen Entwicker bezogener Pakete.

BeOS Auf meiner Installation (BeOS 5 Personal Edition) sind sowohl binutils als auch GNU make mit dem Entwickler Paket gekommen.

OS/2 Ich gebezu ich weiß es nicht. Da sind EMX und native Versionen, und auch DOS ebenfalls. Soweit ich weiß, ?FPC used to be EMX based, but is currently gearing towards native?. Ich muß mich erst selbst schlau machen :-)

Dos/Go32V2 Benutzt DJGPP Go32V2 Werkzeuge. Das ist durch den go32v2 extender den der FPC benutzt bedingt und für das gefarhllose Zusammenhängen von Programmen mittels Extender, muß überall der gleiche Extender verwendet werden. Inklusive der Werkzeuge :-)

Netware Ich habe keine Idee. Ich habe niemals einen Port benutzbar gesehen.

win32/win64 Benutze das mingw Set und bevorzugt welches mit der letzen FPC Veröffentlichung vertrieben wurde. Siehe weiter unten.

wince Meine Erfahrung ist beschränkt auf CE als crosscompilation Ziel. Einige Personen haben zum Beispiel NAS Boxen welche das bootstrapping von FPCauf einem WinCE basierenden Host erlauben.

Mac OS X kommt mit den Apple Developer Werkzeugen (welche installiert werden mit dem XCode auf 10.3). Die Benutzung von allgemeinen Unix Bibliotheken wie mysql, ncurses, postgresql und so weiter, kann FINK erfordern.

Die Situation auf Win32 verwirrt oft die Leute. Das wird dadurch hervorgerufen, das es zumindest drei Sätze von den obigen Werkzeugen (ar,ld,as,make) gibt, die auf Windows laufen. Ein mischen (Ein Werkzeug von dem einen Satz, eines aus einem anderen) kann in unvohersagbaren Ergebnissen enden. Wie auch immer, die drei Kandiaten sind:

The situation on win32 often confuses people. This mainly because there are at least three available sets of the above utils (ar,ld,as,make) that run on Windows. Any mixing (one util from one category, one from the other) can also lead to unpredictable results. Anyway, the three candidates are:

[3] welches unter fpc/utils/fpcm/fpcmake.ini gefunden werden kann

Mingw (Auch mingw32 genannt) welches eines der benutzbaren ist. Ein Win32 spezifischen Werkzeug mit Win32 feeling (Laufwerksbuchstaben, Backslashes im Pfad und vieles mehr). Die bevorzugte Version der FPC Veröffentlichungen, weil sie FPC spezifische kritische Patches enthalten. Zum Beispiel ein wichtiger Punkt ist, das die mingw Werkzeuge nach einem in Großbuchstaben umgewandelten PATH Variablen suchen und nicht nach der wie "Path" geschriebenen. Dieses Verhalten ist mehr verträglich auf einem NT basierenden Windows. Siehe auch im Win32 spezifischen Teil der FAQ auf der FPC Webseite für mehr Information.

Cygwin gibt die maximale kompatibilität mit Unix und kann als Unix kompatibilitäts Layer gesehen werden. FPC, wie auch immer, ist nativ auf Windows und ein Buildsystem das halb Windows und halb Unix ist, ist schwer instand zu halten. Der FPC kann auf einer Cygwin installation erzeugt werden, das Ergebnis ist allerdings kein Cygwin Programm. Weil Cygwin Programme benötigen die cygwin1.dll in der richtigen Version, auch wenn sie nicht für die Ausführung benötigt wird (ausser für die Textmodus IDE). In neuerster Zeit, erweitert Cygwin die Unterstützung der nativen Pfade, wenn diese voll qualifiziert sind. Mingw ist besser, allerding ist Cygwin benutzbar in Notsituationen.

go32v2 Go32v2 ist dos basierend und wird über das DOS Kompatiblitätssystem von Windows benutzt. benutzen sie es nicht mit dem Win32 Compiler, es ist so, als würden sie die Win32 Werkzeuge mit Wine unter Linux nutzen :-)

Ein allgemeines Problem ist, so habe ich es herausgefunden unter Win32, das Cygwin "bin" Verzeichnis, in den Win32 Suchpfad aufzunehmen. Das mingw make.exe findet die Cygwin Shell und versucht einige Programme damit zu benutzen. Cygwin wurde um einiges weiterentwickelt, es sieht aus als würde es wieder arbeiten, wenn alles auf einem Laufwerk ist (Man kann den FPC neu erzeugen, alleine mit Cygwin und dem FPC Kommandozeilencompiler).

Als Hinweis, das ander Entwicklungwerkzeuge (Borland, Microsoft, java) auch ein Paket mit make haben können. Geben sie immer den FPC als das erste Verzeichnis in die PATH Variable.

Durch die neue Möglichkeit des parallelen Kompilierens beim benutzen der FPC make Dateien, **make 3.80 wird dringend Empfohlen ab dem FPC 2.1.1 (Jänner 2007 und später).** Unix Benutzer mit 2 Prozessorkernen und aktuellen Sourcezweigen werden "make -j 2" verwenden wollen.

1.2.5 FPC selbst. ppc386, ppcppc, ppcsparc, fpc.

Free Pascal ist hauptsächlich mit sich selbst geschrieben, den Pascal Dialekt den er selbst unterstützt. Das hat einige Konsequenzen für den Beginn des Selbsterstellen (Bootstrap).

- Ein normales erstellen des FPC ist normalerweise nur machbar mit dem FPC als Startkompiler.
- Bootstrapping von Delpi aus war unter gewissen Umständen möglich, allerdings werden einige Erfahrung vorausgesetzt, weil es die make Datei es nicht unterstützt. Delphi kompatibilität wurde vernachlässigt, weil zu viele Delphi Bugs aufgetreten sind und die Kompilierung zu Problem. Irgendwann zwischen 1.9.4 und 1.9.6 wurden die meisten Delphi workarounds entfernt. D2005 wurde nie getestet. Die Möglichkeit Delphi als Startcompiler zu verwenden ist nicht mehr gegeben weil sich Delphi weit von einer verwendbaren Punkt befindet, so ist es einfacher FPC zu verwenden.
- Bootstrapping von TP (TurboPascal) ist möglich für Versionen für 1.0.x und früher. 1.1.x verwendet Delphi Klassen. ? However already quite some mastership was required for this, since the compiler is a large program, and the single datasegment limitation of TP was severe?.
- Bootstrapping von GNU GPC oder p2c ist nicht machbar. Ihre TP Modies sind nicht TP/BP kompatibel genug und der FPC 1.1.x und später benötigt Delphi Kompatibilität.
- Bootstrapping mittels VP ist in Theory machbar, zumindest mit 1.0.x mit einigen Kenntnissen. Es ist niemals getestet worden, weil der FPC ist besser auf fast allen Plattformen, besonders für die grundlegenden Rekompilierungen und weniger Fehlerbehaftet.

Der Grund für diese Auswahl beruht auf einen ?advocacy document? mehr als hier.[4]

Praktisch hier ist ein großer Hinderungsgrund: Sie benötigen FPC um den FPC zu bauen. Und ein großer Vorteil: FPC benötigt im allgemeinen weniger Werzeuge die installiert werden müssen, im vergleich zu einem GCC erstellen. Man benötigt nur ein benutzbares Binärprogramm um den FPC zu bauen, keine komplette FPC installation (auf Plattformen auf denen die GNU Werkzeuge nicht vorhanden sind, werden auch die benötigt, siehe auch oberhalb) und hier werden **definitiv keine Bibliotheken benötigt.**.

Mit dem interene Linker, ein einiger Zeit, kann die Notwendigkeit der GNU Tools fallen und es erlaubt einen einzigen Compiler der kompletten Bootstrap des ganzen FPC/Lazarus Projektes.

[4] Die FAQ listet einige Gründe auf, aber meines Wisssens nicht alle.

Die datei die sie benötigen um einen Compiler aus einem normalen Snapshot zu erzeuegen ist "ppc<processor>", so ist der ppc386 für x86, ppcppc für den PowerPC. ppcsparc für Sun Sparc V8 Systems, ppcx64 für den x86_64 (64-bit x86) und so weiter. Bis jetzt sind diese Dateien statisch gelinkt. So gibt es nuer einen Linux compiler, einen FreeBSD Kompiler und so weiter. Kernel, Bibliotheken und Distributionsversionen sind nicht tragend (ausser es sind extereme Kerneländerunegn, wie der Wechsel von Linux 1.0.x auf 2.0.x)

Für das Crossbuilding auf dem selben Prozessor, auf einen unterschiedlichen Betriebssystem benötigt keinen speziellen Cross-Compiler. Nur für das Crossbulding zu einem anderen Prozessor wird ein anderer Compiler benötigt.

Nebnebei, da ist ein " **fpc**" Binary. Dieses ist ein gemeinsamer Frontend für alle ppc<cpu> Compiler auf einem System. Einer der für das aktuelle System kompilert und einer der für alles andere kompiliert. Der Grund ist, das man nur ein Binary benötigt und den Prozessor mit -P angeben kann. So erzeugt ein

```
fpc -Ppowerpc compileme.pp
```

erzeugt "compileme.pp" und benutzt den PowerPC (ppcppc) Compiler, wenn alles richtig konfiguriert wurde. Wir sehen uns das später genauer an :-)

Der FPC Compiler kann auch verwendet werden num die Compiler Version zu setzen:

```
fpc -V1.0 compileme.pp
```

wird den standard Compiler(ppc<current processor>) mit dem -1.0 suffixed verwenden. Verbindung mit -P ist möglich. fpc -Ppowerpc -V1.0 wird versuchen das Binary ppcppc-1.0 als den realen Compiler auszuführen. Speziell unter Linux ist das schön, weil es möglich ist nur mit Symlinks die Version einfach zu wechseln.

Kombiniert mit einer meisterlich erzeugten fpc.cfg Datei ist es möglich ein leistungstarkes crosscompiling Systems mit automatisch auswählenden Ziel zu erzeugen. Wir kommen dazu später wie man das macht.

1.2.6 Andere Werkzeuge

Hin und wieder werden andere Plattform abhängige Werzeuge benötigt. Die am meisten verwendeten sind der Windows resourcen Compiler windres um Resourcenscripts (*.rc) zu kompilieren und dlltool um Importbibliotheken für den FPC zu erzeugen, so das andere Compiler (MSVC, mingw) sie benutzen können.

1.3 Quellen bekommen

1.3.1 Einführung

Direkt nach der FPC Veröffentlichung, wechselte das FPC Projekt auf SVN als Quellcodeverwaltungssystem. Die Gründe waren hauptsächlich das Verzweigungs- (branching) und Zusammenführungsfunktionen (merging) des SVN.

Normalerwesie wenn man einen Snapshot baut, wird der Source aus dem SVN geholt und kommt als großer Quellkodezweig (sourcetree) mit /fpc als Wurzelvereichnis (root directory). Die Quellcoden sind auch herunterladbar als Archiv von der Hauptseite [5][6]. Dieses Archiv wird jedes Monat größer und ist derzeit 25MB, bereits komprimiert, groß.

Der beste Weg die Quellen zu holen, ist per SVN, welches auch das Auffrischen (update) auf die aktuelle Version inkremental ermöglicht (in andern Worten, es läd nur die Änderungen herunter).

1.3.2 SVN Modules

Der FPC und Lazarus Quellcode ist über einige SVN Module gestreut:

Module	Description
fpc	Der FPC Hauptteil. Er beinhaltet Compiler, RTL, Textmodus IDE und die meisten nicht visulenne Bibliotheken.
fpcdocs	Die FPC Dokumentationsquellen (in LaTeXund fpdoc XML Format)
fpcprojects	Projekte die derzeit nicht gedacht sind mit dem FPC zusammen verteilt zu werden, sind hier geparkt. (inkludiert IRC bot)
fpcbuild	Beinhaltet den fpc und fpdocs Zweig als auch das Installations-, das Demoverzeichnistree und eine make Datei für Release pa
lazarus	Das Lazarus Projekt (Visual classes library LCL und die Lazarus IDE/RAD)

[5] \href{ftp://ftp.freepascal.org/pub/fpc/snapshot/v24/source/fpc.zip}{ftp://ftp.freepascal.org/pub/fpc/snapshot/v24/source/fpc.zip (2.2.x)}
[6] \href{ftp://ftp.freepascal.org/pub/fpc/snapshot/v25/source/fpc.zip}{ftp://ftp.freepascal.org/pub/fpc/snapshot/v25/source/fpc.zip (2.5.x development series)}

1.3.3 Herunterladen der Quellen mittels SVN : basically retrieving a local working copy from a checkout. Typical reasons to build in an export instead of a checkout are

Um ein Check out eines Teils durchzuführen, verwenden sie die folgenden Zeilen:

```
svn co http://svn.freepascal.org/svn/<module>/trunk <module>
#
# Examples:
#
# FPC
# svn co http://svn.freepascal.org/svn/fpc/trunk fpc
#
# fpcdocs : basically retrieving a local working copy from a checkout. Typical reasons to build in an ex
#
svn co http://svn.freepascal.org/svn/fpcdocs/trunk fpcdocs
#
# fpcprojects (has no branches, maybe you don't need /trunk at the end)
#
svn co http://svn.freepascal.org/svn/fpcprojects/trunk fpcprojects
#
# lazarus
#
svn co http://svn.freepascal.org/svn/lazarus/trunk lazarus
```

Um einen Zweig aus zu checken, ersetzen sie "trunk" in den folgendne Zeilen durch branches/<branch name>. Zum Beispiel, um den Zweig fixes_2_4 aus zu checken:

```
svn co http://svn.freepascal.org/svn/fpc/branches/fixes_2_4 fpc-2.4.x
```

Eine spezielle version wird mit einen Tag (Zeichen) versehen welcher wie folgt formatiert ist- RELEASE_2_4_0 - und kann wie folgt aus gecheckt werden:

```
svn co http://svn.freepascal.org/svn/fpc/tags/RELEASE_2_4_0 fpc-2.4.0
```

1.3.4 Updating der Quellen mittels SVN : basically retrieving a local working copy from a checkout. Typical reasons to build in an export instead of a checkout are

Der Vorteil des holen der Quellen mittels SVN ist das inkrementale auf Stand bringen (incremental updating). Es ist sehr einfach mit SVN, einfach nur "svn up x" verwenden. Wenn x ein Verzeichnis ist, in welches früher mittels SVN etwas aus gecheckt wurde, so werden Http Pfad, Änderungen und Zweige automatisch vom System gelesen.

Beispiel:

```
svn up fpc
#
svn up fpc-2.4.0
```

1.3.5 Reverting SVN checkouts

Das was neu im SVN ist, ist rückgängig machen (reverting). Wenn lokale Änderungen Konflikte mit dem auf Stand bringen hevorruft oder man 100% sicher sein will das es keine lokalen Änderungen gegeben hat, sollten sie die Änderungen rückgängig machen. Damit sichergestellt das ihre lokale Kopie wirklich mit dem SVN Server übereinstimmt.

Sie sollten das tun, wenn sie ein Problem haben, das andere auf dem IRC oder Maillist nicht nachvollziehen können und der Startcompiler ist korrekt (siehe auch Bootstrapping)

Beispiel:

```
svn revert -R fpc
```

1.3.6 Exporting (Win32 Benutzer, lese das hier !)

Export ist Grundlegend, Erzeugen einer lokalen Arbeitskopie von einem Checkout. Der typische Anwendungsfall für einen Export statt einem Checkout sind:

- Sie machen einen Release Build. Um zum Beispiel, RPMs, debs, freebsd ports und so weiter, zu machen
- Sie arbeiten unter Win32 und wollen "make install" verwenden um den Build UND die Beispiele zu installieren.

DasProblem unter Windows ist, das einige Werkzeuge Probleme mit den Read-Only Attributen auf administrativen Dateien von einigen SVN-Clients haben. Der SVN export kopiert nur Dateien aus dem Repository und keine administrativen Dateien, so ist dies ein guter Workaround. Exporten ist einfaches kopieren, so benötigt der Workaround nahezu dieselbe Zeit, als wenn man manuell aus dem Checkout herauskopiert.

Das Format des Exportkommandos ist:

```
svn export path\to\checkout exportdirectory
```

Beispiel:

```
svn export d:\fpc fpcexport
oder
svn export /usr/local/src/fpc fpc
```

Wenn etwas unter dem zweites Argument vorhanden ist, so wird SVN die ausführung ablehen. In diesem Fall kann man das Argument -force verwenden um das überschreiben zu erzwingen.

Note 1: Ich habe es jetzt nicht getestet, aber das Bereinigen des Repositories vor dem Export kann etwas Geschnindigkeit bringen, speziell unter Windows.

Note 2: Für eine **schnellere Lösung** schauen sie in der Build Tricks Sektion nach.

1.3.7 Mehr Information über den SVN

Mehr Informationen über den FPC und den SVN kann unter http://www.freepascal.org/wiki/index.php/SVN_Migration gefunden werden.

1.4 Extension Konventionen

FPC definiert einige Dateierweiterungen, welche eine nähere Erklärung benötigen:

.o Der aktuelle Kode einer kompilierten Unit oder des Hauptprogramms.

.a Der aktuelle Kode einer kompilierten Unit oder des Hauptprogramms wenn Smart gelinkt wird.

.ppu Der Rest der kompilierten Unit, welche keine aktueller Kode ist (wie Information über Direktiven welche beim Kopmpileren definiert werden, das geparste Interface, und so weiter)

.s Assembler Kode (werden zu .o übersetzt) erzeugt vom Kompiler.

.as Assembler Kode in Quellform (wovon es nie ein Pascal equivalent gegeben hat). Meistens programmstarter Kode.

.s Dateien. .rst Resourcestring Dateien für die Internationalisierung.

.$$$ Temporäre Dateien. Kann ohne Gefahr entfernt werden.

.res Windows oder OS/2 Resourcen Datei

link.res Linker Script Datei. Beinhaltet die Information aus welchen Dateien die Binärdatei gemacht wird und welche externen Bibliotheken benutzt werden.

ppas.sh (oder .bat) Batchdatei welche den Linker mit den richtigen Argumenten aufruft.

***.lrs** Lazarus Resources, alte Version.

***.rc** Resource Quelldatei, wird von Windres verwendet um .res Dateien zu erzeugen.

Die Erweiterungen auf Win32 haben ursprüglich in einem "w" (*.ow, *.ppw) geendet, was gemacht wurde um eine Verwirrung des Dos Compilers auf dem selben System zu verhindern. Wie auch immer, es hat sich durch eine bessere Struktur der Verzeichnisse überlebt, die neue Struktur war nötig für leiche Crosscompiling, und in dem 1.1.x/1.9.x Zweig wurden die Dateierweiterungen für Win32 wieder .ppu/.a/.o

So findet man eine .ppu, eine .a und an .o Datei je kompilierter Unit.

1.5 Directory layout

Grundlegend ist das FPC Layout sehr flexibel. Das Binary (ppc386, ppc68k or ppcppc) muß nur fähig sein, die Konfigurationsdatei zu finden und die Datei spezifiziert den Rest. So sind die Layouts die hier beschrieben werden, die standard Layouts der Releases oder für die Zukunft geplant. (letzters natürlich Haupsächlich für das multi Processor Design)

Die grundinstallation des FPC hat folgende Hauptkategorien:

binaries Die Basis Binaries (fpc, ppc68k und ppcppc, ppcsparc, ppcx64) müssen im Suchpfad (PATH) des Betriebssystems sein. Wenn sie FPC ausführen, so muß es dem FPC möglich sein, die anderen (ppcppc, ppc386 etc) zu finden, also durch den Suchpfad (PATH).

fpc.cfg Der Binärdateien muss es möglich sein, die Konfigurationsdatei in einer der Standardlokationen zu finden, welche für das Betriebssystem definiert sind. (Siehe plattformabhängiger Sektion)

units Die Konfigurationsdatei muss den richtigen Pfad, je nach kompilierters CPU, zur RTL und anderen Paketen beinhalten. Die Struktur ist normalerweise baumförmig. Crosskompilierte Units sind ebenfall im (Verzeichnis-)Baum enthalten.

(binutils) Die Binutils (LD,AS,AR) müssen auch im Suchpfad sein, oder die beinhaltenden Verzeichnisse müssen in der fpc.cfg oder auf der Kommandozeile mittels dem -FD Parameter angegeben werden. Auf Betriebssystemen, welche die Binutils standardmässig nicht installieren, werden diese durch die FPC Installation ins selbe Verzeichnis installiert. Dies ist kein Problem, wenn nicht Corsskompiliert wird, oder zwei FPC Installationen gemischt werden (zum Beispiel eine DOS und eine Windows Installation auf einer Maschine). Als Hinweis, das das angeben der fpc.cfg Datei nur arbeitet wenn die Programme per Hand kompiliert werden. Wenn ein neuer Kompiler erzeugt (gebaut) wird, so wird die Konfigurationsdatei fpc.cfg durch das makefile ignoriert.

Nebenher gibt es auchnoch Quellcode Bäume, Dokumentation, die Kompiler Sprachlokalisierung und das beispieleverzeichnis. Diese sind nur für das Erzeugen von echten Distributionsreleases von interesse und beeinflussen das arbeiten des Kompilers selbst nicht. Das kommt daher, das der Kopiler selbst die Quellen in kompilierter Form in den e .o, .a und .ppu Dateien in den units Verzeichnissen hat. Der Platz wo die Sourcen sind ist nicht so wichtig. Der einzige der es für automatisierte Prozesse benötigt, ist die Lazarus (IDE) und dort kann man jeden Pfad mittels dem Lazarus FPC Quellcode Dialog angeben.

Wenn Pfade in der fpc.cfg Datei geschrieben werden, so werden einige Ersetzungen (substitutions) automatisch gemacht, zum Beispiel

$FPCTARGET wird durch die Architektur - Betriebssystem (architecture - operating system) Kombination für welche kompiliert wird ersetzt. (z.B. i386-linux)

$FPCVERSION wird durch die Version des FPC welcher es liest ersetzt.

$FPCCPU wird durch die Angabe des Ziel-Prozessors ersetzt.

$FPCOS is replaced by the operating system.

$FPCFULLVERSION is replaced by a compiler version with an extra patchlevel. (2.4.x)

$FPCDATE is replaced by the current date.

Since filesystems hierachies differ with the OS, I'll put some OS specific information in the following sections.

1.5.1 Unix (and full clones like Linux)

Under Unix, most directories are relative to the so called prefix. A prefix is a kind of root directory for package installing. Common prefixes are /usr, /usr/local, /usr/exp and /usr/pkg (the last one is common on NetBSD).

The usual convention is that programs that come with the base-distribution go into /usr, and hand-compiled or hand-installed packages go to /usr/local. However the definition of base-distribution varies with the distribution and OS. The main idea is to give root (the administrator) trusted users the ability to install and manage less important packages in /usr/local while keeping the base system only writable for the administrator (group) himself. Linux distributions are traditionally more likely to install packages into /usr instead of /usr/local, but not every distribution does that. "man hier" should list some of the conventions used on your Unix version.

If directories are relative to a prefix, they are noted as $PREFIX/the/rest/of/the path. Note that this has nothing to do with $FPC<x> substitutions in fpc.cfg. $PREFIX is shellscript notation for "get content of environmentvariable PREFIX". Compiled units go into $PREFIX/fpc/$FPCVERSION/units/$FPCTARGET. Units used for crosscompiling also go into $PREFIX/fpc/$FPCVERSION/units/$FPCTARGET.

Binaries go into directory $PREFIX/bin, however sometimes these are only symlinks to the actual files in $PREFIX/lib/fpc/$FPCVE directory to easily switch versions. Since the binary can find its own unit files using versioned info in fpc.cfg when properly set up, usually the default binary is the only thing that needs to be swapped.

The place where configuration files are searched on unix are: /etc/fpc.cfg, ~/.fpc.cfg. $PREFIX/etc/fpc.cfg is searched from 1.9.8 onwards. (note; ~ means home directory on Unix)

So let's take $PREFIX=/usr/local, $VERSION=2.4.0, $FPCCPU=i386, $FPCOS=freebsd and $FPCTARGET=i386-freebsd as an example, and see what file goes where:

File(s)	location	remarks
fpc	/usr/local/bin	
ppc386	/usr/local/bin	actually symlinked to /usr/local/lib/fpc/2.4.0/ppc386
ppcppc	/usr/local/bin	actually symlinked to /usr/local/lib/fpc/2.4.0/ppcppc
rtl for i386-freebsd	/usr/local/lib/fpc/2.4.0/units/i386-freebsd/rtl	
rtl for i386-linux	/usr/local/lib/fpc/2.4.0/units/i386-linux/rtl	(cross operating system compiling)
rtl for powerpc-netbsd	/usr/local/lib/fpc/2.4.0/units/powerpc-netbsd/rtl	(cross architecture and OS compiling, endianness)
rtl for x86_64-win64	/usr/local/lib/fpc/2.4.0/units/x86_64-win64/rtl	(cross arch, OS, wordsize)

Note: 1.9-1.9.4 stored libraries for cross compilation in $PREFIX/lib/fpc/$FPCVERSION/cross/$cpu/$target. Because all platforms are now fully qualified with architecture, a separate directory for CPU is no longer necessary, and FPC 1.9.5 and above store everything in $PREFIX/lib/fpc/$FPCVERSION/units and below.

1.5.2 Windows and Dos

The windows and dos case is pretty much the same. Except that $FPCTARGET=i386-win32 for windows, and $FPCTARGET=i386 go32v2 for dos. **It is advised to avoid using spaces in directory names. While (win32) fpc can handle it, some versions of the binutils and other tools can not (yet), or need carefully placed quotes.**

Both these two OSes have a specific directory with all FPC related files (including the documentation etc) in them, default is usually used to be c:\pp, but has been changed to c:\fpc\2.4.0 since 2.0. I call this path $INSTALLDIR. Usually there aren't other versions in the same $INSTALLDIR A different version means a different $INSTALLDIR, at least in the default situation.

All binaries go into $INSTALLDIR\bin\$FPCTARGET. This directory should be in the PATH. This directory is quite full, since there are other tools and utilities installed. However if the target is marked to require 8.3 compatible naming (Go32v2, OS/2), the binary path is not $FPCTARGET (e.g. bin/i386-go32v2) but just $FPCOS (e.g. bin/go32v2).

Compiled units go into $INSTALLDIR\units\$FPCTARGET and deeper, at least under 2.0+.

Configuration files are searched in the same directory as the binary ($INSTALLDIR\bin\$FPCTARGET), but if a environment variable HOME exists also in %HOME%/.fpc.cfg.

So let's assume $INSTALLDIR=c:\fpc\2.4.0 and the default $FPCTARGET=win32

File(s)	location	remarks
fpc	c:\fpc\2.4.0\bin\i386-win32	
ppc386	c:\fpc\2.4.0\bin\i386-win32	
ppcppc	c:\fpc\2.4.0\bin\i386-win32	
rtl for i386-win32	c:\fpc\2.4.0\units\i386-win32\rtl	(cross architecture and OS compiling)
rtl for i386-linux	c:\fpc\2.4.0\units\i386-linux\rtl	(cross operating system compiling)
rtl for powerpc-netbsd	c:\fpc\2.4.0\units\powerpc-netbsd\rtl	
rtl for x86_64-win64	c:\fpc\2.4.0\units\x86_64-win64/\rtl	
Cross binutils	c:\fpc\2.4.0\cross	(not installed by default setup)
libs for i386-win32	c:\fpc\2.4.0\libs\i386-win32	(not installed by default setup)
libs for i386-linux	c:\fpc\2.4.0\libs\i386-linux	(not installed by default setup)

1.5.3 Where are my units?

Starting with later versions of FPC 1.9.5, the fpc makefiles create a directory to store the created units. The main reason for this is building for multiple targets without cleaning in between.

So when just build (not installed), the rtl units files are in fpc/rtl/freebsd/units/$FPCTARGET. This is a silly example of course, since the RTL is platform specific, but this change is systematic, so for the RTL too.

1.6 The fpc.cfg configuration file

Note: Former versions used ppc386.cfg as configuration file. Even latere 1.0.x versions already support fpc.cfg. In time ppc386.cfg will be dropped, so *please* use fpc.cfg.

A correct configuration file should at least do the follow things:

1. (**most important**) Provide the place where to find the correct precompiled units for the current selected operating system and architecture. In IDE's this option is usually called UNITPATH Option: **-Fu**
2. If other binutils then for the default platform need to be used (like in the case of crosscompiling), then fpc.cfg should allow the compiler to find them. I don't know what name IDE's typically use for this, but BINUTILS PATH or GNU UTILS PATH would be appopriate Option: **-FD**
3. Extra paths where static and shared libraries are searched. Often called library path Option: **-Fl**
4. (minor) By default the compiler shows only very little errors, warnings and information. It's convenient to add some verbosity parameters to fpc.cfg. Option **-vihw -l**. [7]
5. (crosscompiling) Binutils for cross compiling are often prefix with cpu-operatingsystem(e.g. i686-ming32),this prefix can be specified using -XP<prefix> parameter, or using makefiles using BINUTILSPREFIX. Option **-XP**

There are other interesting possibilities of fpc.cfg, (like internationalised errormessages, setting the compiler in a different mode per default, always to try smartlinking etc) but these are the important ones for normal development.

Note: Keep in mind that when using the makefiles supplied with the FPC source, the fpc.cfg is **NOT** read. This is to avoid problems with two RTLs, the old and the new one, when bootstrapping the system.

1.6.1 Unit path -Fu

The -Fu case is pretty much as described in the directory layout paragraphs (1.5), there are two things in which the -Fu compiler option differs from the directory layout as described before. If I append an asterisk (*) to a path, it means that it should search all directories below that directory too, and that the layout for crosscompiled units is a bit different.

We can use the directive FPC_CROSSCOMPILING to detect crosscompilation. The FPC manual has a list of other defines that can be tested.

Unix: If we assume that the compiler is installed to PREFIX /usr/local, and we are using FreeBSD/i386 as host OS, this becomes:

[7] The warning system is currently being expanded to have more flexibility.

```
# This is pretty generic for OSes with *nix filesystem layout
-Fu/usr/local/lib/fpc/$FPCVERSION/units/$FPCtarget/*
#....or suitable for crosscompiling. Note that the /cross/ dir is not necessary
# and only added as example.
#ifdef FPC_CROSSCOMPILING
# OS not default -> CROSS
-Fu/usr/local/lib/fpc/$FPCversion/cross/$fpctarget/units/*
#else
 #ifndef cpu86
 # Processor not default -> CROSS
  -Fu/usr/local/lib/fpc/$FPCversion/cross/$fpctarget/units/*
 #else
  -Fu/usr/local/lib/fpc/$FPCVERSION/units/$FPCtarget/*
 #endif
#endif
```

Win32 and Dos installed in c:\fpc\2.4.0

```
-Fuc:\fpc\2.4.0\units\$FPCtarget\*
# or suitable for crosscompiling
# OS not default -> CROSS
-Fuc:\fpc\2.4.0\units\$fpctarget\*
# win32 binutils are in the path
#ifndef win32
# set path to crossutils. Assuming in one dir.
-FDc:\fpc\2.4.0\bin\cross
# this is not 100% safe. GNU and FPC target naming differ
-XP$FPCTARGET-
#endif
```

Keep in mind that extra paths (e.g. for own custom packages) can be added. There is also no reason not to use $FPCVERSION in the win32 case. It is just an example.

1.6.2 Binutils path -FD

The problems with binutils

1. We only have to set -FD if we set up for crosscompiling. Either when cross compiling to a different processor or a different OS. So we have to determine default OS and CPU somehow.
2. Under Unix, there are usually specific dirs for crossutils, but the exact place and name differs with version. Win32 doesn't have something like that at all. For win32 I propose an example layout as described above.
3. FPC and GNU platform naming differ

Unix : As an example I take FreeBSD on an ordinary PC. The location below (/usr/local/processor-operatingsystem) is where the binutils base distribution installs cross compiled utils by default under FreeBSD. However sometimes how FPC names an OS can differ from how the binutils name it. In that case you have to be more verbose, which I have done for the case of crosscompiling to windows[8] .

So our configuration file becomes:

```
#ifdef FPC_CROSSCOMPILING
# other binutils if OS differs
#ifdef win32
# win32 is an exception
# FPC OS name : win32 Binutils OS name : mingw32
# FPC processor: i386 Binutils processor: i686
```

[8]Lazy people simply would make a symlink from the fpc naming to the binutils naming. However being lazy is not allowed for tutorial writers.

```
-FD/usr/local/i686-unknown-mingw32/bin
#else # we hope that the fpc and binutils name match:-)
-FD/usr/local/$FPCTARGET/cross
#endif
#else
#ifndef cpu86
# other binutils if processor differs
-FD/usr/local/$FPCTARGET/cross
#endif #endif
```

Win32 and Dos: Pretty much the same. However contrary to Unix I haven't really tested this. Again I assume FPC to be installed in c:\fpc\2.4.0 .However the binutils are in d:\binutils and deeper and named like under *BSD:

```
#ifndef win32
# other binutils if OS differs
-FDd:\binutils\$FPCTARGET\bin
#else
#ifndef cpu86
# other binutils if processor differs
-FDd:\binutils\$FPCTARGET\bin
#endif
#endif
```

Using makefiles

When using makefiles, one can set the -FD parameter by passing CROSSBINDIR=<path> to make. This ensures that the parameter is passed correctly from makefile to makefile (in nested directory hierarchies).

1.6.3 Binutils prefix, all binutils in one directory

A somewhat newer addition is the -XP parameter which sets a prefix for all binutils, so if you add **-XPbla-die-bla-** to the fpc commandline then all calls to as and ld will be prefixed with **bla-die-bla-** . The main reason for this is that by default, cross compiled binutils are named <cpu>-<target>-<filename>,and not just <filename>.

When doing a cross snapshot (or cycle), setting the make variable BINUTILSPREFIX will make sure that -XP is passed on the right moments.

If we assume the binutils for cross compiling are all in one directory, the above piece of code becomes:

```
#ifdef FPC_CROSSCOMPILING
 # other binutils directory if processor or target OS differs
 -FD/usr/local/cross/bin
 # set prefix to select the right ones from that directory:
 -XP$FPCTARGET-
 #endif
```

This assumes that binutils naming of platforms is the same as FPC's. It isn't however, there are a few exceptions:

Usual name	FPC name	Binutils name	ppc<x> name
(32-bit windows)	win32	cygwin or mingw32	(OS not CPU)
i386 or x86	i386	i386,i486,i586,i686	386
Sunos/Solaris	sunos	solaris	
PowerPC	powerpc	powerpc	ppc

So for these targets we would have to make exceptions. (see e.g. script code in the fpcbuild/install repository, e.g. samplecfg and install.sh)

1.6.4 Library path -Fl

The library path is where static libraries (and shared libs under unix) can be found. While this is very important for the Unix platform, the FPC compiler on windows and dos also can link to GCC (and variants) libraries, so FPC must be able to find them.

The library path is of course target and processor dependant. One couldn't link FreeBSD/x86 libraries to a Linux/Powerpc binary. So essentially we are doing the same trick as with the unit path. If the default targets are used, on unix it is set to the default directories, otherwise it is set to some hierarchy where the user is supposed to install his cross libraries.

Unix : As an example I take FreeBSD on an ordinary PC, because most BSDs, specially on ordinary PCs have a "compability" Linux libraryset to run proprietary Linux programs for which no FreeBSD version exists. This mode is called the Linuxator[9]. The Linuxator libs are usually SUSE based and in /compat/linux/ and deeper. NetBSD has a huge set of such emulations. Anyway, here is the snippet, it assumes the user to build a hierarchy of cross libs under /usr/local/crosslibs

```
#undef specialoscpu
#ifndef FPC_CROSSCOMPILING
#define specialoscpu
# DEFAULT
# libraries outside base are installed with PREFIX=/usr/local
-Fl/usr/local/lib
# however X stuff is installed with PREFIX=/usr/X11R6
-Fl/usr/X11R6/lib
#endif
#ifdef linux
#ifdef cpu86
# CROSS, BUT SPECIAL, FreeBSD optionally has a Linux userland on board
# for the linuxator. Usually it is some SUSE version.
#define specialoscpu
-Fl/compat/linux/lib
-Fl/compat/linux/usr/lib
-Fl/compat/linux/usr/X11R6/lib
#endif
#endif
# libraries not existing on target by default. Reverting
# to special cross directories
#ifndef specialoscpu
-Fl/usr/local/crosslibs/$FPCTARGET/lib
-Fl/usr/local/crosslibs/$FPCTARGET/usr/lib
-Fl/usr/local/crosslibs/$FPCTARGET/usr/local/lib
-Fl/usr/local/crosslibs/$FPCTARGET/usr/X11R6/lib
#endif
```

Win32 and Dos: Exactly the same idea. Some directory with the hierarchy (d:\crosslibs for now), but none as default. If you have a suitable gcc equivalent (djgpp for dos, cygwin and mingw for win32), you could add it to the space for the default.

```
#ifndef win32
# other binutils if OS differs
-Fld:\crosslibs\$FPCTARGET\lib
-Fld:\crosslibs\$FPCTARGET\usr\lib
#etc
#else
#ifndef cpu86
# other binutils if processor differs
# maybe somebody does a port for win32/Sparc or so.
-Fld:\crosslibs\$FPCTARGET\lib
-Fld:\crosslibs\$FPCTARGET\usr\lib
#else
```

[9] The linuxator is strictly speaking an emulator. However the emulation layer is so thin (a few 10's of kbs) that there is no noticable performance loss. Since all libs are in memory twice (Linux+FreeBSD) it eats some extra memory though.

```
# DEFAULT
#endif
#endif
```

1.6.5 Verbosity options

Well, there is not much to tell. The default is ok, and I never change it (I wrote it down below for completeness), for more info see the manual. Note that any verbosity setting in fpc.cfg can be overriden on the commandline (e.g. for debugging, typically uses -va on the cmdline)

```
# write FPC logo lines.
-l
# Display Info, Warnings, Notes and Hints
-viwn
```

1.7 The order to compile parts of FPC

Packages is a bit of an ambiguous term when used in the FPC project[10]. Sometimes "packages" refers exclusively to the separate unit sets in the packages/ directory of the source tree, sometimes it also includes the other separately compilable parts of the project (like compiler, rtl, fcl, ide, fvision, lazarus, installer and utils). In this paragraph it's the latter one.

The directory structure of the packages has been reorganized starting with FPC 2.2.4. [11]. Most importantly, the fpcmake system now figures out its own dependancies, so all packages (including FCL and FV) could move to /fpc/packages. This great simplifies building and maintenance of the packages. The units of the FCL were also regrouped into multiple packages.

Of course there iss till a certain sequence one should follow when building FPC, which is hardcoded in the top Makefile. This because some packages need other packages and need to be build first. The usual order is:

1. bootstrap the compiler (make cycle) 3x compiler, 3x rtl
2. Rebuild the RTL smartlinking
3. Build the packages directory including the FCL and FV packages.
4. Build the textmode IDE.
5. utils/ directory including fpcmake in utils/fpcm

Lazarus can be built after the main build (with or without IDE). However install the main build *first*. Some directories are not built at all in a standard build of the fpc repository:

1. demoes and example directories of packages
2. the regression testing suite (tests/)

Examples are often compilable by descending into the package directory and executing "make examples".

1.7.1 Some interesting build scripts

Some of these principles are used in some of the scripts used to build releases, test crosscompiling etc.

- fpc/installl/install.sh is the installer script for unix. Nicely demonstrates platform name (GNU->FPC) conversion
- fpc/install/cross/buildcrossbinutils is a script to mass-build cross binutils
- fpc/install/cross/buildcrosssnapshot is an attempt at building cross snapshots. Works in principle, however the shared linking of mkxmlrpc and the textmode IDE are troublesome.
- fpc/install/makepack is a release building script. It requires the target name (i386-linux, i386-win32 etc) as parameter

[10] See alsohttp://wiki.freepascal.org/packages `http://wiki.freepascal.org/packages`

[11] This reorganization was made possible with improvements in the fpcmake system. However in time, the fpcmake system will disappear, and be replaced with the fpmake/fppkg system that is already included for testing in 2.2.4+

1.8 Smart linking

Smartlinking (ld/gcc documentation calls it dead code elimination) essentially means avoiding linking of unused code into the final binary.

The problem is that the GNU linker can do dead code elimination, but usually (read: with gcc) only does this on a compilation unit level. In other words, it still will link in the entire unit when only one file is used.

Newer binutils use can do deadcode elimination based on sections, however the problem with this is that it ups the version requirements on binutils, which is a problem specially for operating systems as classic MacOS, older Mac OS X versions, OS/2 etc. The new form is also not entirely stable yet.

Using the old style smartlinking, FPC circumvents this by generating a .o for each symbol in a unit. So each procedure, method, variable or typed constant gets its own .o file. Since this can be thousands of .o files in a unit (unit Windows has about 20000-30000),and the .o files are added to an .a archive using GNU AR.

All together this is a pretty complicated process:

1. a unit is compiled by fpc to a lot of assembler source (.s) files,
2. that get assembled by a lot of calls to AS to generate a lot of .o files
3. that are read again by AR to form one .a file

.... which is why smartinking performance really benefits from the binwriter (see II), which causes FPC to write the .a directly without thousands of calls to AS.

Besides tightening code, smartlinking also solves a problem when a unit interfaces to a shared library (unix .so or win32 .dll) and certain unused calls aren't available. An example would be if unit windows contains calls added in later Windows versions that you don't use for your program and you work on/for Windows 98. when not smartlinking, the whole of unit windows would be linked in, and the linker wouldn't be able to find the calls from later Windows versions in the Windows 98 kernel32.dll and user32.dll.

Be careful when trying to guess the savings of smartlinking. Often people forget unit initialisation (both of the unit in question, and the unit it USES) and all the classes, procedures and functions the initialisation uses are always linked in, and units (even if they are unused) are always initialised, since the compiler can't see that the effect of the initialisation on the working of the program is zero.

A word of caution: In C, an .o file usually corresponds to one .c or .cpp file, so one .a contains <n> .o's that correspond to <n> .c's. However under FPC, when smartlinking per unit (.pp , .pas) the file that is ultimately written is an .a which contains a lot of .o's. So an FPC .a corresponds to one unit only. [12] An .o however is the same unit as the .a without smartlinking.

A known problem with old style smartlinking is its **memory usage**. Ld doesn't manage a lot (100000+) of small object files (.o) efficiently, and memory requirements increase enormously. Smartlinking a fully static Lazarus wasn't possible because it bombed out on my OS' 512 MB memory limit for ordinary processes, a rough estimate of the memory needed was 1.5 GB. A rule of thumb for memory usage in extreme cases is between 50 and 100 times the size of the final binary.

It seems that ld can do deadcode elimination nowadays, however the compiler needs to be adopted for this, and there is also some movement in having an own internal linker. The advantages of this system is less memory use by LD, and the fact that .a files will disappear (in the new method, the smartlink data is added to the .o files). However no matter how fast this becomes usable, old style smartlinking is still needed for quite a while, for OSes with binutils that are not recent or complete.

[12] At least for now, developments in library generation might change this in time, but not before 2.4

Chapter 2

Standard building

The start of the FPC buildprocess is bootstrapping a compiler. Bootstrapping the compiler is the start of every build process, because changes in the compiler source must be actived before one can start compiling the libraries. Usually the bootstrap is only a minor part in an automised snapshot or full build. However some knowledge about the process can be handy for crosscompiling purposes and development on FPC itself.

The other two typical build processes are building and installing a snapshot, and building a release. Somewhat less streamlined are building the IDE and documentation. Building Lazarus on the other hand is fairly easy.

For the average user, building a snapshot and/or lazarus are the interesting parts.

2.1 Cleaning

2.1.1 Make clean

In nearly any place in the FPC tree a "make clean" will try to remove compiler generated files of snapshot or cycle building. A "make distclean" in the upper level is even a bit more thorough, and also removes files generated while building releases (zips etc)

Note that the make files usually delete what they generate. So if e.g. the generation of certain files depends on certain makefile defines and/or compiler version, make sure that the right compiler and defines are passed to make.

A good check to see if the repository is clean, is to check standard output and -error of the svn update process. If you see errors, then revert the SVN checkout.

2.1.2 make distclean

make distclean is the more rigorous way of cleaning. If you experiencing build problems, always do this first. If you have snapshot scripts, add make distclean in the top level dir as a first line. This became a lot more important in 1.9.5, since recent 1.9.5's store their units in a separate units/$fpctarget hierarchy. Cleaning before SVN update can speed up the SVN update process significantly, specially on OSes that have relatively slow directory searching (like windows and netbsd)

2.1.3 Cleaning of crosscompile builds

Note that the clean targets of the makefile only clean for current $FPCTARGET. To rigorously clean everything, kill all .o, .a and .ppu files, and then SVN update again, to recover the few .o files in the FPC tree (mostly for testsuite tests relating to C linking)

2.2 Bootstrapping: make cycle

"make cycle" is a bootstrap of the compiler, and builds the rtl and compiler at least three times.

The only starting compiler that is guaranteed to work is the most recent release compiler for that series[1]. So if you have a problem in any aspect of building the FPC source tree and you are not using the most recent release compiler, *please try first* with the most recent release compiler. It is possible to specify the name of the starting compiler (see below), so saving the release compiler under some name (I'll use *ppcrel* from now on in this tutorial) is advised.

The make cycle process compiles RTL and compiler three times:

RTL 1st Since the we start with only the starting compiler, a new RTL must be built first before we can link a program.

Compiler 1st A new compiler is created. However the new compiler is still compiled by the release compiler, so newer optimisation and changes aren't active yet in this binary. For that reason, both the RTL and the compiler have to be rebuilt.

RTL 2nd So we rebuild the RTL and...

Compiler 2nd the compiler. However to make sure the compiler is deterministic (compiles the same program to the same executable each time it is run), the compiler and RTL are compiled.....

RTL 3rd again....

Compiler 3rd and after building the compiler for the third time the 2nd and 3rd compiler are compared to see if there they are the same. If not, everything is compiled again. (for some rare secundary effects)

This entire process is run by simply type "make cycle" in the compiler directory. If you want to use an alternative compiler, specify it with PP=[full/path/to/]<compiler>. When using the PP= option, use a compiler in the PATH or an absolute path, not a relative path. Using a relative path usually won't work since during the process, the make utility changes directories. If you try to bootstrap without a full instaltion, you sometimes need to specify FPC=[full/path/to/]<compiler>

Additional compiler options can be added using OPT="<the options>", often used options are turning on debugginginformation with lineinformation (-gl), and increase the verbosity level (-va). However since -va produces a _lot_ of output, it is wise to pipe it to file.

Some examples: [2]

```
# go to the right directory on unix
cd /fpc/fpc/compiler
# go to the right directory on windows
cd /d d:\repo\fpc\compiler
# do an ordinary make cycle:
make cycle
# Use a starting compiler in the OS search PATH
make cycle PP=ppcrel
# use a starting compiler with (unix) absolute path
# and enable line info and debugging
make cycle PP=/fpc/startcompilers/ppcrel OPT="--gl"
# Use LOTS of verbosity in the process and redirect it to file.
make cycle OPT="-va" >cycle.log
```

2.3 Compiling a snapshot; make all or make build

"Make all" is the most important build process, it essentially builds (but not installs) the general purpose parts of the standard SVN tree (see 1.7), except sideways related stuff like documentation, examples and tests. Installing is one additional makefile command.

In its simplest form, build a snapshot is checking out SVN (see 1.3.3), and changing to the top fpc/ directory and call "make all". Installing to the default location is done by executing "make install".

```
cd /fpc/fpc make all
# lots of building. Think 3 minutes on a XP2000+, 10-15 minutes on a 500MHz machine.
make install
```

[1] E.g. For the 1.9.x series, even though 1.9.6 and 1.9.8 will probably work, the only guaranteed starting compiler is 1.0.10!
[2] The lines starting with "#" in the example below are comments. Don't enter them

```
# Installs the snapshot to the default location
# (/usr/local/lib/fpc/$FPCVERSION and deeper probably)
#
# or on Windows:
cd /d d:\repo\fpc
make all
make install COPYTREE=echo
# (default is c:\pp\ and deeper)
```

If you track FPC SVN very closely, this is probably what you'll do quite often. However in some special cases, more parameters are needed. Essentially this is building a snapshot. If you trick "make install" into installing in a different directory, the contents of that directory are essentially the snapshot.

The simplest variation is a different starting compiler. This can be done by simply entering PP=<compiler> or PP=/full/path/to/<
after the "make all" prompt:

```
cd /fpc/devel/fpc
make all PP=/my/starting/compiler
# lots of building. Think 3 minutes on a XP2000+, 10-15 minutes on a 500MHz machine.
make install
# Installs the snapshot to the default location
# (/usr/local/lib/fpc/$FPCVERSION and typically)
#
# or on Windows:
cd /d c:\repo\fpc
make all PP=ppcrel
make install COPYTREE=echo
# (default is c:\pp\ and deeper)
```

A standard confusing detail when installing a snapshot is that the target directory (e.g. /usr/local/lib/fpc/2.4.0) depends on the version number. If you would follow the above sequence, and the generated compiler (say version 2.5.1) is not equal to the starting compiler (say version 2.4.0), then make install will install into the wrong directory. The same goes for windows (c:\fpc\2.4.0 ...) This can be solved by setting the compiler that is generated in the "make all" step as a starting compiler to the "make install" step. This will ensure that the "make install" sequence gets the correct versioning info. Since we can't use relative paths, this must be a full path:

```
cd /fpc/devel/fpc
make all PP=/my/starting/compiler/path/ppc386-someversion
#
# Now install using the NEW compiler for the correct version info:
make install PP=/fpc/devel/fpc/compiler/ppc386
#
# Installs the snapshot to the default location
# (/usr/local/lib/fpc/$FPCVERSION and deeper probably)
#
# and for windows:
cd /d c:\repo\fpc
make all PP=c:\fpc\startingcompilers\ppc386-someversion
#
# and installing use the NEW compiler
make install PP=c:\repository\fpc\compiler\ppc386
```

Another very common variation is installing into a different base directory ($PREFIX), this is done by setting a variable INSTALL_PREFIX to the desired directory:

```
cd /fpc/devel/fpc
make all PP=/my/starting/compiler
#
# Stack installs packages that don't come with the package system in /usr/exp
#
```

```
make install INSTALL_PREFIX=/usr/exp
#
# Or on Windows:
# My FPC directory is on d:\pp
#
cd /d c:\repo\fpc
make all PP=ppcrel
make install INSTALL_PREFIX=d:\pp
#
# Or lets make a snapshot to upload to some site on some unix:
#
#
cd /fpc/devel/fpc
make all PP=/my/starting/compiler
#
# Make a scratch directory in /tmp
#
mkdir /tmp/fpc
make install INSTALL_PREFIX=/tmp/fpc
#
# archive
#
cd /tmp/fpc
tar cvzf /tmp/newsnapshot.tar.gz *
#  remove temp dir.
#
cd /tmp
rm -rf fpc
#
```

Finally, adding parameters to every compiler commandline is interesting too. This can be done by adding OPT='<the parameters>' to the commandline. Often used parameters for this purpose are -d<xxx> to define a value which can be tested using e.g. {$IFDEF xxx} in the source, and -gl to include debug info. -va is also often used to debug the snapshot building process. If you add -va, pipe the output to file, or the building will slow down due to the massive amounts of output written to screen.

Adding -gl when you are developping with FPC is a good habit, since it allows you to single step all of the FPC runtime libraries in GDB (roughly equivalent to the "use debug .DCUs" options in Delphi). The generated snapshot gets somewhat larger though.

```
cd /fpc/devel/fpc
make all OPT='-gl'
# lots of building.
make install
# Installs the snapshot to the default location
# (/usr/local/lib/fpc/$FPCVERSION and deeper probably)
#
# or on Windows:
cd c:\cvs\devel\fpc
make all OPT='-va' &> d:\buildlog.log
make install
#
# multiple parameters are passed like this:
#
#
make all OPT=''-gl -va''
```

A word of caution: When I build snapshots for home use, I simply "make install" them over the old snapshot, at least if "make all" succeeds. If you encounter errors during build or install, or even after install (like the notorious "can't find unit xxx", while it is there), try to remove the units/ directory in the FPC homedir, and re-execute the "make install" used to install it. This should resolve the problem. If not check as much settings as you can check and read chapter 4

The problem is usually caused by units changing directories (or packages) in CVS. Since the usual procedure doesn't clean the units/ directory, after installation two instances of that unit exist, which could lead to conflicts.

A good habit is to backup source tree before updating and the FPC install directory (compiler + units/ directory) before installing a new snapshot. This just to make sure you will have a working compiler in case something goes wrong during building or installing. This can easily be automated using a script that updates - backups - builds - installs. Since I'm not that fond of unix scripting, I'll leave that as an exercise to the reader.

2.4 Current build procedure on win32

The situation on win32 with current SVN is a bit complex, so I decided to dedicate a separate paragraph listing the procedure, combining all the above data. The problem on win32 is that SVN markes its own config files as read-only, which leads to problems with installing examples, and the cp utils provided with 2.0 have some unix path <-> windows paths troubles. Most notable, they don't see \xxx\yyy as a windows path, but they do see d:\xxx\yyy as a windows path.

There are two solutions, the quick one simply omits the examples, the slow one does. Best is to do the slow one once in a while to have up to date examples, and for the rest use the quick solution. The slow solution also fixes some problems with older versions of the mingw utils.

Besides the read-only svn dirs problem, there is also a problem with Vista, and some mingw binaries have problems with the casing of PATH statements.

2.4.1 The slow solution: SVN exporting

Before we begin, lets define the directories (my defaults)

- d:\pp is the directory where I currently have installed FPC, and d:\pp\bin\i386-win32is the FPC dir in my %PATH%.
- d:\fpcSVN is the directory where I checked out the SVN repository. I don't build in it.
- d:\fpc is the directory where I export the SVN repository too.
- d:\fpcrel is the directory containing FPC 2.4.0 for bootstrapping purposes.

Another important issue: **MAKE ABSOLUTELY SURE THAT THE CYGWIN DIRECTORY IS NOT IN THE PATH!!!!!!** The reason for this is that some mingw utils start using cygwin sh.exe when found, and that in turn swaps some utils from the FPC delivered mingw tools to cygwin ones.

Then, to make and install a snapshot, we update the SVN:

```
cd /d d:\
svn up fpcSVN
```

Then we export it:

```
# Cleaning improves performance,
cd fpc
make clean
cd ..
svn export --force fpcSVN fpc
```

Now, the building can start:

```
cd fpc
make clean all OPT='-gl' FPC=d:\fpcrel\bin\i386-win32\ppc386.exe
```

If this is successful we install: (all on one line)

```
make install OPT='-gl' INSTALL_PREFIX=d:\pp11 UPXPROG=echo
                       FPC=d:\fpcrel\bin\i386-win32\ppc386
```

Some things to notice:

- **MAKE ABSOLUTELY SURE THAT THE CYGWIN DIRECTORY IS NOT IN THE PATH!!!!!!** Mingw/msys can also give problems, but more rarely (see 2.6.4)
- The options (OPT='-gl') are passed both to the make and install lines. This because sometimes due to unit dependancy glitches and utils, a minor amount of code is (re) built during install. This goes for **ALL** build options that are not installing related
- UPXPROG=echo avoids UPXing of the resulting binaries, so that the FPC binaries can be debugged. If you want as small FPC bins as possible, leave it away, and add SMART=1 to both make lines. Note that UPXing wastes memory that is more expensive than disk. [3]
- We add –force to the SVN export line to force overwriting of the previous repository. If building goes wrong, day after day, try erasing the export directory (d:\fpc) before exporting.

2.4.2 The Quick solution: COPYTREE

The quick solution exploits that the recursive copying is done using a command that is *only* used for copying the examples. And updating the (installed) examples is not that important when daily updating, one typically wants to avoid the slow SVN export step. The makefile macro that is used internally in the makefile for copying files recursively is COPYTREE, we then use the same trick as earlier to disable functionality by specifying the "echo" command:

```
make install COPYTREE=echo
```

2.4.3 VISTA specific topics: install.exe

Vista is quite draconian in some things, and there will be a lot more problems in the future, specially for release engineering. (think about temporary files in the root, not allowing writes in the appdir etc). However there is one little gotcha that also affects normal building.

This gotcha is actually the result of an attempt to minimalize trouble for existing installers by automatically popping up UAC (otherwise you would have to do "run as admininstrator" which is not end user friendly), but it was implemented half-assed. Any EXE with install, setup or update in its name will pop up UAC, but.... *UAC prompts fail for console apps* .[4]

The FPC build system uses GNU install (as ginstall.exe) to install files with proper permissions, and that fails due to UAC.

The definition of the problem also implies the solution:

1. Copy the (g)install.exe binary to a new name without the keywords, I use "myinst.exe"
2. Pass GINSTALL=myinst.exe to the make commandline :

```
make install GINSTALL=myinst.exe
```

P.s. This Vista/Windows 7 problem is fixed in 2.4.0 by adding a manifest.

2.4.4 Putting it all together

Here is the batchfile that I use on Windows: (the FPC source tree is in d:\repo\fpc)

```
@echo on
set BASEDRV=c:
set SRCDIR=%BASEDRV%\repo\fpc
set PPCNAME=ppc386
set FPCSTART=c:\fpc\2.4.0\bin\i386-win32\%PPCNAME%
```

[3] http://wiki.freepascal.org/Size_matters`http://wiki.freepascal.org/Size_Matters`

[4] http://technet2.microsoft.com/WindowsVista/en/library/00d04415-2b2f-422c-b70e-b18ff918c2811033.mspx `http://technet2.microsoft.com/WindowsVista/en/library/00d04415-2b2f-422c-b70e-b18ff918c2811033.mspx` search for "keyword"

```
set LOGDIR=%BASEDRV%\repo
set INSTALLDIR=%BASEDRV%\pp11
REM some random opts.
set OPTS=-gl -dSAX_HTML_DEBUG -dUSE_MINGW_GDB
set COMMONOPTS=UPXPROG=echo COPYTREE=echo OPT="%OPTS%" GINSTALL=myinst.exe
rem === invariant part ===
cd /d %SRCDIR%
REM the building
make clean all %COMMONOPTS% FPC=%FPCSTART% 1> %LOGDIR%\buildlog.txt 2>&1
REM separate install step for crossversion purposes (and under Unix sudo) (on one line)
make install %COMMONOPTS% INSTALL_PREFIX=%INSTALLDIR% FPC=%SRCDIR%/compiler/%PPCNAME%
                                1> %LOGDIR%\installlog.txt  2>&1
```

Actually, I have several such files, another one for 64-bit, a more advanced one for crosscompiling to wince etc. The win64 is a copy that differs only in "x86_64-win64" in the path and ppcx64 instead of ppc386. Note that all variables are in the upper part of the batchfile, this makes it easier to port it to systems and just adapt the paths. Having several computers with slightly different path and drive layouts was the motivation to write them.

Unfortunately, pure batchfile is too limited to invest too much time in it.

2.5 Lazarus

Building lazarus is actually pretty much the same as building a snapshot, except that you have to enter directory lazarus/ instead of fpc/.

```
cd /repo/
svn update lazarus
cd lazarus
make all
# you might want to check if the return value of ‘‘make install’’ is zero before installing.
make install INSTALL_PREFIX=/usr/local
#
```

At the moment of writing this there was some problem with installing lazarus. It seems that the makefiles aren't entirely up to date. I usually move the lazarus repository to /usr/local/lazarus, and set some symlinks to execute the right binary.This isn't that bad, since you need to save the lazarus source for Lazarus' operation anyway.

2.6 Typical problems and tips

2.6.1 cannot find -l<xxxx>

Since Lazarus need to link to shared libraries, building lazarus tests an aspect of the FPC configuration that hasn't been tested by the parts above. A typical error is

```
Linking ./lazarus
/usr/libexec/elf/ld: cannot find -lglib12
lazarus.pp(334) Error: Error while linking
Closing script ./ppas.sh
```

This means that the linker, when building the final binary, couldn't find libglib12.a or libglib12.so. This is officially not an FPC error, since glib is part of GTK, which belongs to the operating system. *So maybe the gtk (or a corresponding gtk-devel package) is simply not installed.* There are four other possibilities:

1. While the package is installed, the package manger didn't create a symlink that refers the libglib.12.so.x to libglib12.so. A quick cd /usr/local/lib; ln -s libglib12.so.2 libglib12.so would have solved that in this case.

2. (the actual problem in the case above) The directory with the correct file isn't searched for libraries by FPC, because you didn't specify the directory in fpc.cfg using -Fl. A mere adding of -Fl/usr/local/lib to /etc/fpc.cfg solved the problem in this case.
3. You have a {$LINKLIB xx} in the source somewhere, or a -l<xx> parameter in some script or makefile that forces FPC to try to link the library, but it isn't necessary anymore, grep is your best friend in such cases.
4. Your distribution for some reason names it libraries differently. Again, symlinking the searched name to the real name is the solution.

2.6.2 CONTNRS or other FCL-BASE units not found

If the compiler can't find contnrs, this points to a problem with the -Fu paths. It probably means -Fu was wrong, but the makefiles managed to auto-add the RTL, and the contnrs (and other FCL-BASE) unit(s) are then the first unit that can't be found. It can also point to duplicate .ppu files or mixing of FPC versions.

2.6.3 Piping stderr and stdout to the same file.

Piping stderr and stdout of the MAKE process to the same file is useful for debugging the exact sequence on events. This is easily done on Unix using >&.

Under Windows this is a bit more complex, but can still be done:

```
make install 1> filename.txt 2>&1
```

This line pipes the output of install's stdout to filename.txt using 1>, but redirects stderr (2>) to stdout (&1) first.

2.6.4 (Windows) building and install fails with "invisible" errors

I suddenly had problems when doing "make clean all install" in one go. Further investigation of the output showed "ignored errors" in make exit lines:

```
make: [fpc_clean] Error 1 (ignored)
```

however this would lead to an all stop after the *all* part:

```
make: *** [fpcmade.i386-win32] Error 1
```

This turned out to be a mingw related problem, a file "sh.exe" was left in the project source. Another solution would be to never use these commands in one go, but script them as separate ones.

I experimented with sh.exe because when using "make -j 2" it produces a warning that -j functionality is disabled if sh.exe is not found. I assume the makefile must be adapted to ignore "ignored errors" instead of "all errors" somewhere.

Note: Despite the usage of mingw tools, it is not recommended to have the mingw directories in the path. Some of the mingw tools (most notably make.exe when it detects sh.exe) apparantly change behaviour upon detection .

Chapter 3

Crosscompiling

(you need to have read the previous chapters too, since they already contain a lot of crosscompiling related info that I won't duplicate here)

3.1 Basic crosscompiling of a snapshot

Let's have two examples, freebsd to win32-mingw crosscompile, and a FreeBSD to AMD64 linux one.

Assume we have the following items set up correctly:

1. Cross binutils have been compiled, and are installed with $PREFIX=~/cross. The correct prefix has been identified (probably something like i686-ming32-and x86-64-linux-in our example)
2. The FPC sources are in fpc/
3. FPC was installed in /usr/local/lib/fpc/$FPCVERSION and deeper

Then we execute:

```
cd ~/fpc
gmake distclean
# next all on one line
gmake all install OS_TARGET=win32 CROSSBINDIR=~/cross/bin BINUTILSPREFIX=i686-ming32-
                                          INSTALL_PREFIX=/usr/local
```

to build the first snapshot (note that CPU_TARGET isn't set, and that OS_TARGET uses the FPC platform naming (win32), while BINUTILSPREFIX uses the GNU one (since it is for the binutils). Also note that the BINUTILSPREFIX ends with a dash. This is on purpose.

Similarly, for x86_64-linux the line becomes

```
cd ~/fpc
gmake distclean
# next all on one line
gmake all install CPU_TARGET=x86_64 OS_TARGET=linux CROSSBINDIR=~/cross/bin BINUTILSPREFIX=x86_64-linux-
                                          INSTALL_PREFIX=/usr/local
```

3.2 Crosscompiling without cross-assembler.

Sometimes one doesn't want to really crosscompile, just test if something like the RTL compiles under target <x>. If this target is on the same architecture, one can try to use

```
gmake fpc_units OPT='-s'
```

The -s skips calling the assembler/linker and the use to the fpc_units makefile-target skips the assembling of the RTL's loadercode. Another such trick is to pass AS=echo or AS=gecho to gmake. Note that in former (2.4.x/4.x) times, crosscompiling between Linux/x86 and FreeBSD/x86 was possible without cross assemblers and linkers. This to my knowledge hasn't been retested with newer versions of these OSes.

3.3 Crosscompiling Lazarus

3.3.1 Crosscompiling Lazarus from windows to Linux

First, we need a set of cross binutils. For windows->other platforms, these are on FPC FTP `ftp://freepascal.stack.nl/pub/fpc/contrib/cross/mingw/binutils-2.15-win32-i386-linux.zip` . Download and extract them to the same directory as your "ppc386" compiler binary. Verify their install by running i386-linux-ld on the command prompt.

Second, we need FPC and Lazarus source trees. Note that the FPC tree must be an exported tree, since we are going to use make install, and make install trips over SVN directories. So after checkout svn export the sources (see SVN paragraph for examples)

Now we are going to build and install FPC for crosscompiling (assume FPC is installed in c:\fpc\2.4.0)

```
make clean make OS_TARGET=linux all make OS_TARGET=linux install INSTALL_PREFIX=c:\fpc\2.4.0
```

Lazarus is a different matter however, since it uses shared libraries. So I copied Linux libraries from my target system and put them in d:\linuxlib. Which libraries I copied is described in the separate paragraph below.

Before we really start, we have to workaround a different problem. The compiler still has a detection for pre-glibc systems which is easy to trip if you make a mistake. So go into i386-linux rtl directory (probably c:\fpc\2.4.0\units\i386-linux\rtl) and copy cprt21.o over cprt0.o so that both files are in fact cprt21.o.

Now we start with the Lazarus building, enter the lazarus directory and give a

```
rem on one line:
make OS_TARGET=linux all OPT="-gl -Fld:\fpc\linuxlib -Xr/usr/lib -FL/usr/lib/ld-linux.so.2
                          -XLAc=c,dl,gmodule"
```

The -gl is about adding debuginfo (good to have a traceback if something goes wrong), the Fl argument specifies the directory to look for the linux libraries, and the -Xr argument makes the linker create the binary so that it searches in /usr/lib for libraries. The -FL argument is the name of the dynamic linker, and the -XLA line adds two libraries (dl and gmodule) if libc is included.[1]

This should build a Linux lazarus. However most likely, it will bomb out missing some library. The solutions to that are editing the linker file "link.res" and rerunning ppas.sh again, or adapting the d:\fpc\linuxlib dir with more or renamed libraries. The link alias options (as -XLA above) are really handy to avoid repeated editing.

3.3.2 Preparing a directory with Linux libraries

For the crosscompile I copied a bunch of files of the target linux distribution (and yes, these are possibly distribution and version dependant!). Some of these are also needed for the textmode IDE. If library files were named libname.so.x.y on linux I renamed them to libname.so in my Windows directory, since the symlinks used for that on Linux are not easily copied.

Most of these libs are in directories like /lib, /usr/lib and maybe /usr/local/lib, /usr/X11R6/lib and /opt/gnome/lib. libgcc however is in a GCC install dir, which can be found by doing a gcc -v and then look for a line like

```
"Reading specs from /usr/lib/gcc-lib/i486-linux/3.3.5/specs"
```

then some of the libs are in /usr/lib/gcc-lib/i486-linux/3.3.5/ Anyway, these are the files I copied (FC4 iirc):

```
libpthread.so.0
libdl.so
libc.so
ld-linux.so.2
crtbegin.o
crtbeginS.o
crtbeginT.o
crtend.o
```

[1] Alternately, you can add {$linklib dl} and {$linklib gmodule} to the main lazarus sourcefile lazarus.pp

```
crtendS.o
crtn.o
crti.o
libgcc.a
libX11.so
libXi.so
libglib-1.2.so
libgmodule-1.2.so.0
libgdk_pixbuf.so
libgdk-1.2.so
libgtk-1.2.so
libXext.so
libm.so
libdl.so.2
libgmodule-1.2.so
```

Note that some directories are duplicate, with a suffix (like libgmodule-1.2.0)and not. I need them twice because some of the other libraries names the full name (so the form lib<name>.so.x) as an dependancy and we can't symlink on windows, so I simply copy it.

Making mistakes with renaming is not that bad, there will be chances to fix it. Make sure all crt* and a file "libc.so" are available or generating link.res will go wrong. In my case compilation for step 11 will go ok, but the linker will complain it can't find libgtk.so and because on the target system, libgtk is gtk 2.0, while we want gtk1.2 for lazarus (<=0.9.26) which is named libgtk-1.2. I solve this with the so called linkordering switch -XLAgtk=gtk-1.2 and similarly for gdk and glib. See the separate paragraph about this switch and its limitations for more info.

3.4 Interesting compiler and makefile options

3.4.1 Skipping built in directories: -Xd

On Unix targets, Free Pascal often automatically passes default libraries like /lib and /usr/lib as include directories to the linker. This can be a problem when experimenting with crosscompiling (and dual architecture) systems. The parameter -Xd convinces the compiler to not add these directories.

3.4.2 Difference in paths: -Xr<directory>

-Xr configures where the linker should look for libraries when crosscompiling. It is translates to the ld –rpath-link parameter.

3.4.3 -Xt static linking

The -Xt parameter tries to link static as much as possible by passing -static to LD, the linker. A more finely grained control over what libraries are linked statically, and which ones are linked dynamically can at this time only be achieved by postediting the linker "link.res" file. Compile with -s, edit link.res, and then run ppas.bat or .sh to resume the linking process.

3.4.4 CROSSOPT

The Makefile of compiler/ allows to specify compiler options that are only used during the actual crosscompiling state using CROSSOPT= (so not during the initial bootstrap cycle that is not cross-)

3.4.5 LIBDIR

Allows to specify a dir to be used for linking. Don't know the exact cross-compile consequences, specially if the host OS needs a lib path to cycle (Solaris, OS X)

Chapter 4

Systematic problem solving for the FPC build proces

The questions of people that read the previous "make cycle faq" on the FPC maillists suggested that the main remaining problem is what to do if something goes wrong, and to determine the exact cause of the problem.

The usual problems are:

1. Wrong versions of FPC for a certain job or trying to combine compiler and rtl that don't belong together (2.3 with the 2.2.x RTL, or trying to generate windows binaries with the go32v2 binutils). Specially Lazarus users often try to build with a too old version anyway, despite the minimal version advise on the main Lazarus site.
2. Leftovers of the previous install/build frustrate the current build process (not deleting old directories etc, stale .ppu's or .o's) Be hygienic with respect to old .ppu, .o's, binaries and buildtrees!
3. Omitting details that don't have to be done for each install, (not fixing the symlink in /usr/local/bin/ that points to the real installation)
4. Directory inclusion problems (fpc.cfg inclusive). Directories not being searched, or wrong ones included
5. (windows) Having cygwin in the PATH, which causes fatal mixing of the mingw FPC build tools and their cygwin counterparts. Same for other development packages that e.g. provide "make"
6. (unix and strictly not a FPC problem) not installing operation system parts and libraries. Most notably the -devel libraries on Linux, or ports on *BSD.
7. (development versions only) Trying to compile something in a directory that contains files with the same name as files in the FPC cvs. The compiler find these, and thinks they are the RTL sources that need recompilation. Release versions are protected against this (by compiling the RTL with -Ur)

Double checking that you are not having these specific common problems is worth the effort.

4.1 Determining the problem, increasing the verbosity.

The standard systematic search for the problem starts with increasing the verbosity level, by passing OPT=<switches> after make. This works when executing the commandline compiler too.

The commonly used verbosity options are

-vt (show filesearching information, searched paths etc) If you expect problems with what directories are searched, you usually choose this to keep the amount of logged data down.

-va (show all info. REDIRECT THIS!)

This is the ultimate source if something goes wrong. The rest is pretty much guess work and experience. The verbose compiler messages are the best source by far, and the only one that allows some systematic problem searching. Nobody expects the Spanish Inquisition, but try to answer these questions for yourself when looking at the -va output.

- Is the correct compiler executed?
- Does the compiler version match with the version that you'd expect ? (You can also use ppc386 -i to check version). This includes verifying the compiler date!
- Are the operating system and processor you are compiling for correctly named in the output?
- Is the correct fpc.cfg (name and location) loaded and is it preprocessed as you think?
- Are linker, include and unit directories correct?
- If the compiler can't find a unit, does a line like "unit changed, recompiling" exist a bit higher? If so see the separate section on the "recompiling" problem.
- If you are using FPC supplied makefiles, keep in mind that the fpc.cfg file is ignored while building a "cycle" or a larger target that depends on cycle (like "make all") Add parameters to the make commandline, set environment variables, or fix config files (e.g. /etc/ld.so.conf if it is a linker directory problem) to fix this.

If you do use nested includefiles, is the nesting the same way as you expect? (a small filter program can be quite handy to "graph" how includefiles are included for complex files)

4.2 Checklist

Some other things easily checked without extra verbosity :

- (on unix) Check the compiler location with "which ppc386".
- Check the compiler version and date with ppc386 -i
- Look at your path variable (echo $PATH on unix, echo %PATH% on dos/windows), and check that the FPC directory for the target you choose is first. Specially make sure that
  1. there is no cygwin directory in the path at all (cygwin tools use non-dos path layout, and need special handling. FPC's mingw based versions of make and the binutils don't mix well with cygwin's)
  2. (Windows) there is no other directory of a development tool (Delphi, JBuilder, VC++) in your path. These can contain own versions of certain tools, mainly make that are not compatible. A make -v or make –version can be useful too.
  3. the PATH variable is spelled in all capital letters. If you have this problem, correct it in the windows environment dialogue (one of the tabs of System settings) A batchfile that can fix this on the go for if you don't have permissions to fix it directly on your work system is
     set a=%PATH%
     set Path=
     set PATH=%A%
     set a=
  4. Check your fpc.cfg. I know, you never edit that one, and it has worked always, but do so anyway. Quickly inserted "I have to get this working now!" changes are easily forgotten.

4.3 Recompiling

A standard gotcha for FPC newbies is when the compiler says it can't find unit xxx, while you are really 100% sure it is there, and the directory with that file is searched. (note : make 100% sure that this is the case, before continuing)

The most common causes are:

1. (by far the most common) stale .o and .ppu's, or incorrect settings that include wrong directories. (with .ppu and .o's). Run make distclean
2. Including the main source trees into your unit searchpath. This is wrong, they should only be in your debugger/codetools sourcepath
3. includefiles and unit names that are not unique.

4. A difference in the defines.

If so, if you do the same build with OPT='-va' and search for "Recompiling"

[Unfinished]

Chapter 5

Misc topics

5.1 Programming models.

Over the last few years, FPC has started to support an increasing amount of platforms that use the Unix specific RTL parts.

Some form of reorganisation was necessary, and because more processor platforms are in preparation for FPC, I decided to look into portability issues, specially creating 64-bit clean code, and alignment and endianness. The main reasons for researching portability for Object Pascal were Free Pascal's plans to port to 64-bits platforms like x86_64, Sparc64 and PowerPC G5.

The term 64-bit clean is somewhat misleading, because it is not limited to 64-bit architectures alone, but a general principle for writing code in a way that it runs on a lot of processors. The basic principle is to only allow conversion between a certain integer type and pointer, and always have a certain integer type (PtrInt) that is guaranteed to scale with pointer size.

In C, the standard "long", or a new "long long" type is used for this. Since pointers aren't generally regarded as "signed", it seems logical to pick an unsigned type to be compatible with a pointer. Signed or unsigned doesn't much matter for compability with C as long as integers are coded using two's complement, and because the interface between C and another language is untyped.

The reason to define a new type Long Long is simple, to allow legacy code that assumes that integer and long are equal in size to keep on working without modification.

A chosen set of types is refered to in C faqs as a "programmingmodel", and are named ILP-x, LLP-x or LP-x with x the processor's wordsize in bits, typically 32 or 64. ILP-x means that Integer, Long and Pointer are x bits wide, LP-x that Long and Pointer are x bits wide, and LLP that Long Long and Pointer are x bits wide.

Some typical programming models are listed in the table below.

Ctype	objpas	Size in model				
		ILP32	LP32	LP64	ILP64	LLP64
char	byte/char/smallint	8	8	8	8	8
short	shortint	16	16	16	16	16
int	integer/longint	32	16	32	64	32
long		32	32	64	64	64
long long		32	-(32)	-(64)	-(64)	64
pointer	pointer types	32	32	64	64	64

- Some points in the table need explaining:
- Standard programming model for current (intel family) PC processors is ILP32. All 32-bits OSes use it.
- The problem with long in C was that there are both large codebases that expect pointer and long to have the same size, while there are also large codebases that expect int and long to be the same size. The compability model LLP64 was designed to keep the long<->int compability by introducing a new type to remain compatible with pointer (long long)
- long long doesn't exist except in LLP64, and both long and longlong don't have an clear equivalent in Object Pascal yet. A new type would have to be created just like under C. Since Pascal has no long<-> ptr equivalence established yet, the long vs long long discussion is unimported. It was however decided to create both a signed (PtrInt) and an unsigned (PtrUint) type that scale with pointer size

- LP32 is used as model for the win-16 APIs of Windows 3.1
- Most *nixes use LP64, mainly to conserve memory space compared to ILP64, since using full 64-bit ints is hardly necessary and a waste of memory. Examples: 64-bit Linux, FreeBSD,NetBSD,OpenBSD
- Win64 uses the LLP64 model, which was also known as P64. This model conserves type compability between long and int, but loses type compability between long and pointer types. Any cast between a pointer and an existing type requires modification.
- ILP64 is the easiest model to work with, since it retains all compability with the much used ILP32 model, except specific assumptions that the core types are 32-bit. However this model is quite memory-hungry,and both code and data size significantly increase. Typically used by commercial unices, like IBM AIX.
- There are no dos models in the table, since dos doesn't have a flat memory model. (a pointer can't be represented by a single value in these models. DJGPP extender models are ILP32 though.

5.2 Link ordering

Starting with 2.0.4 some switches were added that simplify dealing with changing library names. These switches are **BETA** and may change or **disappear** at any time, e.g. when a linker description language is implemented. They were mostly added to be able inventorize how useful such a system would be in practice, mostly it is meant to deal with changes in library names and keep a release running without repackaging it before a final decision is made. They are BETA also because the internal linker might totally change the linking backend in the near future.

The main problem it solves is that with FPC, names of libraries can be hardcoded in the FPC source in **$linklib** or **EXTERNAL** declaractions. While this is generally a good thing that makes it way easier for the user and avoids kludges like having to specify each and every library on the commandline like gcc, it can be annoying in times when librarynames go through a transition, and you use a distribution that renames libraries.

Besides the name, also the order of libraries might sometimes be a problem, specially when libraries don't specify their dependancies correctly. Most of these issues are with relative lowlevel libs like libgcc.

FPC 2.0.4 introduces several **beta** switches to deal with this. There are three new parameters:

1. -XLAlibname=[libname1][,libname2..]
2. -XLOlibname=<number>
3. -XLD

A limitation that remains is that there is no way to allow the user to decide which libraries are to be linked statically, and which are to be linked dynamically. This mainly for easier deployment on linux (non standard libs? -> force static). Another limitation recently surfaced is that this only goes for libraries. Object files, and their order are untouched

5.2.1 -XLA

The first parameter, -XLA, defines a subsitution of library name. The libs on the right side are added to the parameters of the linker if the lib on the left side is in the list with libraries to link that is constructed from the source. The right side can be empty, and can also contain the libname that is already specified on the left. Examples:

- -XLAc=c,dl makes sure that libdl is added if libc is in the list, libc remains in the list.
- -XLAdl= removes libdl from the list of libraries to link even if it was {$linklib xx}'ed in the source
- -XLAgtk-12=gtk12translates a link to libgtk12 to search for libgtk-12

5.2.2 -XLO

The parameter XLO defines a weight. Libraries are sorted on weight into descending order. Defaults weight is 1000. If you decrease the weight, it will be lower in the list with libraries to link. This is mainly useful to link certain libraries (typically the more lowlevel ones) in a certain order, like e.g. with libc_r and libc on FreeBSD4, or to get programs to link that made a mess of their dependancies. This possibility removes the need for a few ad-hoc ordering hacks, and the special parameter for the FreeBSD 4->5 transition that was one such kludge, -Xf was reimplemented using this functionality. The compiler has default weights on a per target basis. This mainly means that libc usually will be somewhere near the end.

Example:

```
-XLOc=1050
```

puts libc before libraries that don't have a weight specified in link.res (since the default is 1000)

5.2.3 -XLD

The parameter -XLD simply makes the compiler forget the defaults for weights and aliases. This can be useful if something changes for a library that has a hardcoded weight, or when a built-in alias is not needed anymore.

5.2.4 Example: Fixing the FreeBSD GTK breakage with linkordering

The FreeBSD ports maintainers decided to force large changes related to linking using libtool on the ports tree in november 2005 (in UPDATING), but they were not into effect till April 2006. One of sideeffects of these changes were renaming of the GTK 1 libraries to gtk-12, gdk-12 and glib-12, where they used to be gtk12, gdk12 and glib12. These changes affected all users that tracked 5 and 6 stable, since they didn't wait for a major version transition to introduce this. Ports maintainers also refused to add a few symlinks for a while to ease the transitions, and pointed to gcc specific tools like gtkconfig and libtool. (these emit gcc commandlines).

Most users track STABLE by default out of practical purposes, and changing the FPC binary to call external programs and to parse them, would only make the whole process more fragile and harder to support. I myself decided to give up working on FPC ports tree entries because I really start wondering where I benefited from participating in the ports tree process. The work is immense, and the uses for the users are small if such changes break packages so hard and relatively often.

Since in time, more and more users will use either -STABLE or a release based on -STABLE after the change, in FPC 2.2.0 I changed the default name of the GTK libraries to their new names, and introduced linkordering switches to keep older installations working. So if you have a system from before this change, and don't track stable, you can workaroundthis by putting the following in your fpc.cfg or .fpc.cfg:

- -XLAglib-12=glib12
- -XLAgdk-12=gdk12
- -XLAgtk-12=gtk12

5.2.5 Example II: FreeBSD 4 vs 5 pthreads:-Xf

Another item is the ppc<x> parameter -Xf, which signals an alternate pthreads layout. Currently it only exists for FreeBSD. On versions 2.0 and 2.0.2 the parameter signals FreeBSD 5.x libpthread behaviour, and on 2.0.4 it signals 4.x libc_r behaviour. In 2.0.4 the parameter was reimplemented on top of the linkordering routines, so that when compiling for 4.x libc_r can be used , [1] en when compiling for FreeBSD 5.x libpthread will be linked in. Also the ordering is fixed using internal equivalents of -XLO, and the defaults are also overridable with -XLD.

[1] 2.0.4 doesn't provide a out of the box 4.x setup anymore, though generally the functionality should be preserved.

Part II

Glossary

Note that this glossary tries to explain compiler and FPC project specific jargon, not Pascal/Delphi in general.

architecture Roughly the processor family. Note that for PC's, the x86 architecture (all 80x86 chips) is often misused, where the i386 architecture (80386 and higher) is meant. The PC version of FPC is i386 architecture, and doesn't run on older x86 proc essors. Other architectures are PowerPC (which is subdivided into power32 (=32-bits) and power64 (=64-bit)), m68k (68030+ in practice), Sparc V7, Sparc V8, Sparc V9 (the latter better known as Ultra Sparc). ARM versioning is more complex, since those processors can be built to order, a customer can choose from several cores, and add modules like FPU, memorysystem (endianness), DSP functionality etc.

AR The GNU archiver. Combines .o files into .a files. .a files are used for smartlinking, though that might change in the near future.

AS The GNU assembler. The backend assembler that is used if FPC needs an external assembler.

assembler a textual representation of the binary machinecode that is fed to the processor.

alias A function or variable with a name that doesn't adhere to standard name mangling, usually for internal functions or foreign language interfacing.

binwriter The part of FPC that directly writes .a's and .o's. Since using the binwriter eliminates executing AS (and possibly AR), building is much faster. The reason why it is not available for each platform is that there is no universal .o format. Every binary type often has its own .o variant, and all need special support in FPC.

branch(CVS,SVN) Normally versions of a file in CVS are a linear series of updates, a new version after each update. This can be thought of as the trunk of a tree. However at a certain point you might want to add an update that might e.g. break backwards compability or that introduces instability for some time. The solution then is to have two sequences of versions. One with that update (a development series of update), and one without. These separate series are the branches of a tree.

bootstrapping is the entire process of creating a compiler starting from another compiler (another version or a totally different compiler). See also **make cycle**

Carbon A Mac OS widgetset/API, which is essentially the cleansed subset of the classic Mac OS API that works with Mac OS X.

CLX A library from Borland that is a bit more multiplatform and should have replaced VCL. However users stayed away in droves. CLX is GPLed, and thus is not very usable for FPC, library wise.

COCOA A Mac OS X widgetset/API written for use with Objective C.

contrib directory A directory in SVN (module fpcprojects) where some ported Delphi packages are kept (ICS, Abbrevia, Decal). This directory is a subdirectory of the projects directory.

Cross (cross -compiling, -linking and -assembling) Generating binaries or libraries for other operating system and/or processor than that you are currently working on.

(crossbinutils) binutils (ld,as,ar) that generate binaries for other operating systems and processors.

(crossbindir) Place where the crossbinutils can be found.

CVS A version management tool used by the FPC tool to manage the FPC source code before may 2005

Cygwin `www.cygwin.com` A distribution of unix tools for windows that tries to stay as close to the original Unix semantics. See also **mingw**

dead code elimination GCC or maybe general C jargon for smartlinking. The default dead code elimination of gcc is not as finely grained as FPC smartlinking though.

ELF binary format of most modern unices and unix likes. Except for Mac OS X (see Mach-O)

Export (libraries) Making symbols (procedures, variables) available to the outside. (e.g. outside the unit, outside the library/program) (SVN)

FCL Free Component Libraries. Non visual class libraries, provided for partial Delphi source compability, at least at a non visual/GUI level. See also **VCL**.

FPC
1. Free Pascal Compiler abbrev.
2. The fpc compiler frontend binary.

FPCMAKE util to convert Makefile.fpc to Makefile

FPMAKE The succesor to fpcmake and make/makefiles in general that is supposed to render the latter obsolete in 2.2. The main features are a more maintainable and performant build system (crosscompiling inclusive)

FPPKG Another build tool scheduled for 2.2 which should allow auto download+compile of packages, a bit like what the FreeBSD ports tree with a apt-get like cmdline frontend would look like. (or portupgrade, which is the proper FreeBSD pendant)

FPDOC A documentation generation system used by FPC internally, but also usable for external use. fpdoc takes a XML file generated by makeskel, and postedited with fpde, processes the file and generates multiple documentation formats from it (html,text,T E X)

GO32v2 Pure Dos is 16-bit. FPC programs are 32-bit. The program that hides the 16-bit nature a bit, and does 32-bit DPMI dos memory management for 32-bits applications is called the Dos-Extender. FPC uses the GNU Go32 version 2 dos extender. (though others are possible in theory, like pmode. See manual)

Inline If procedure X calls procedure Y, and Y is flagged for inlining, then Y will be totally optimised away, and in the final code (in the binary), the code of Y will appear to be in X. If Y is called on two separate places in X, the code will appear twice in X. While nearly anything can be flagged " inline;", some kinds of procedures/methods are excluded from inlining, like virtual methods.

Internal linker A linker built in to the compiler, as opposed to an external linker like GNU LD. FPC uses an internal linker in FPC 2.2+ for the three windows platforms (win32/win64/wince)

Lazarus A multi widget GUI RAD for Free Pascal. Not a blind Delphi drop in replacement, but the support of GTK makes it useful. At the moment of writing this, Lazarus is developing rather fast, for up to date information, see their site Lazarus site

LCL The set of Visual classes used by Lazarus and the programs it generates. The LCL is generally independant of platform or GUI widget library. Starting with 0.9.28 on *nix the LCLwill use GTK2, win32/64/ce on Windows, and Carbon on OS X. A QT based port is also available for *nix.

LD The GNU linker, which is nearly always the one used by FPC.

Mach-O binary format of OS X/Darwin make cycle A command to rebuild the FPC compiler and core RTL from source. (bootstrap, building a new compiler from an older, or other compiler)

make all A command to rebuild the entire base FPC sourcecode. This is roughly equivalent to the snapshots.

mangling see namemangling

mingw (mingw32,mingw64 depending on win32 or win64) A distribution of Unix tools for Windows that adapts more to "the windows way" than **cygwin.**

Namemangling Languages can have different namespaces that can contain the same name, however on the linker level there is only one namespace. Namemangling is the process that maps the multiple language level namespaces onto the single linker level namespace. An example is e.g. the unitsystem. Units can contain variables with the same name. Say unit A contains variable "buildfaq", and unit B does too. Naming both "buildfaq" would make them clash on the linker level. A solution is to prefix the variable name with the unitname. So "buildfaq" in unit A will be called A_buildfaq, and in unit B B_buildfaq. Namemangling is compiler (and -version) specific, and since the namespace rules vary with the language also somewhat language specific. An exception is the C (not C++) language which barely has namespaces, and therefore often has little or no mangling. The C++ namemangling is a notorious case, since it changes from compiler to compiler and version to version.

NEWRA The new improved register allocator of the 1.1.x (2.0+) series. The new register allocator should improve compiler performance and maintainability by automatically arranging a fairly optimal register usage. Since this removes the need for manually keeping track of registers in the codegenerator,this improves maintainability a lot.

Packages is a bit of an ambiguous term when used in the FPC project. (see also http://wiki.freepascal.org/packages `http://wiki.freepascal.org/packages`)

1. Sometimes "packages" refers exclusively to the separate unit sets in the packages/ directory of the source tree,
2. sometimes it also includes the other separately compilable parts of the project (like compiler, rtl, fcl, ide, fvision, lazarus, installer and utils),

3. a third description is the Delphi language feature with the same name, a more automated form of dynamic libraries. In this document it's usually the second one.

PIC (Position Independant Code) a system for Unix (*BSD, Linux) that ensures relocatability of shared libraries, and allows easy minor version upgrades of libc libraries without recompiling the library.

Prefix
1. (PREFIX) place where the snapshot building (make all install) will use as root directory for the installation of the files. (default: \pp on windows, /usr/local on most unices) The dollar sign that is sometimes prepended is the (Unix) notation for environment or makefile variable, a bit like Windows/DOS %something%.
2. (BINUTILSPREFIX) A string that is prefixed to the binutils when crosscompiling. Usually in the form processor-operatingsystem-(like "i686-mingw32-")

Projects directory a directory in CVS where experimental and independant projects are stored. Usually Delphi code ported to FPC (see contrib directory), and Lazarus.

Ptrint A integer type that has the same size (in bits) as the pointer type.

PtrUint is the unsigned form. Register allocator The engine that tries to make optimal use of the processor registers to avoid reloading from memory.

Register parameters A calling convention that tries to put parameters to procedures in registers. Delphi has a register calling convention by default. FPC is slowly starting to support register parameters now (thanks to NEWRA changes). Most non PC processortypes also use register parameters. Since december 2003, FPC defaults to this convention on i386 too, for Delphi compability.Duringthe last months of 2006, a bug was found in the Delphi compability of register parameters; when procedures use more than 4 parameters (one less for methods), the fact that FPC pushes the parameters in the opposite order of Delphi becomes a problem. This probably will be tackled in FPC 2.1.1 in spring 2007.

Register variables An optimization technique that avoids reloading variables from memory into the main CPU's registers by trying to keep frequently used variables in registers as much as possible. FPC 1.0.x had this in a primitive way based on reserving a few registers for variables. FPC 2.x has a way more intelligent system as a part of the registerallcoater

RTL RunTime libraries. The base set of libraries used by FPC. These include e.g. the System unit, the dos unit, the sysutils unit and a platform dependant unit. The System unit also contains functions that the compiler calls internally to do more complex language operations like SET and string handling.

shared linking Linking with dynamic libraries (unix: .so shared libraries; win32: .dll shared libraries)

smart linking essentially avoids linking of unused code into the final binary. See the separate paragraph (1.8) elsewhere in this document for more details.

static linking The ordinary way of linking, producing one standalone binary, though sometimes smartlinking is also called "static" linking. This might be true for C, but FPC has two different modes because with FPC the granularity of smartlinking typically is higher.

Starting compiler A compiler that is used to build a new compiler.

Subversion(SVN) The current version management system.

tag(CVS,SVN) A tag simply identifies a bunch of different files that belong together. It's commonly used to mark a the set of versions that are released as a formal (pre)release. E.g. the FPC 1.0.10 release is tagged with RELEASE_1_0_10 etc.

target
1. Roughly the operating system and CPU that the binary must run on, typically in CPU-OS notation (e.g. i386-freebsd) There are exceptions though, like Dos has a target for each extender, and some compilers have multiple targets for Windows (gcc: mingw, cygwin) too. Some toolchain let the CPU reflect the CPU that is optimised for (i386,i486, i586 etc), some not. FPC does not (all i<x>86 platforms, x>=3, are called i386)
2. Lazarus adds the widget set too, e.g. x86-win32-win32 (i386+ processor, win32 OS, and win32 (GDI) widgetset) and x86_64-win64-win32 (means 64-bit x86 CPU, win64 OS, and win32 widgetset (GDI)).[2]

VCL Borland's classes libraries. Since these contain bot win32 GUI and non-GUI classes. the VCL rougly equates to the non visual FCL classes plus the visual classes in Lazarus' LCL.

Widgetset A library or API for visual output. (originates on Unix where core drawing, and windowing is separated)

[2]The fact that lazarus names the widgetset for win64 "win32" doesn't mean it is some form of 32-bit emulation. The differences were simply too small to define an additional target

win32
1. The API of Windows 9x and NT,2000,XP that is roughly the same over all versions. (though newer versions add more calls). A cut down win32 api is available for win3.1x and is called "win32s". FPC doesn't support win32s (actually: it was never tested is a better description)
2. The FPC target for this api
3. In Lazarus it is also the designation for the graphical part of the win32 api (which is actually called GDI, roughly the widgetset). Lazarus windows binaries for target win32 and win64, have as widget set.

Win64 The 64-bit variant of the win32 API. Very close to win32.

wince The windows api the mobile Windows versions (wince/pocketpc/windowsmobile) are based on. Note that nowadays also "XP embedded" exists which is win32 based, but based on the NT kernel, so specially with tablet PCs make sure what you exactly have.

winNT The common underpinnings of NT4, windows 2000 (NT5) ,XP (NT5.1) ,2003 (NT5.2),Vista (NT6), 2008 and 7 (NT7)

windres A resourcecompiler used to turn resource scripts into .res files that can be included by the linker.

Index