Какво е TMOS?

TMOS като съкращение е Thumb Mode OS и представлява операционна система, но като проект съдържа и много концепции, които са извън рамките на една операционна система.

Поддържани таргети

TMOS е предназначен за ARM микроконтролери, основно Cortex M3/M4. Като драйвери се поддържат за SAM series (atmel), LM series (luminary, cera Stellaris TI) и STM32 (ST).

Необходими инструменти

 TMOS е GCC базиран проект. Няма изисквания за определен тулчейн, но се препоръчва yagarto тъй като е най-изчистен, актуализира се редовно и е мултиплатформен. Последното е много важно, тъй като разработката на TMOS проекти трябва да може да става както под Линукс, така и под Windows.

За IDE се препоръчва Еклипс. За дебъг може да се използва всеки gdb-съвместим дебъгер, но за jtag/swd емулатор по възможност се препоръчва такъв дето поддържа трейс съответно по dcc/swo.

Концепция за компилация и организация

 Организацията на проектите може да е по-сложна и да има повече тънкости, отколкото една ОС. Целта на използваната концепция е да бъде максимално гъвкава, но без да е прекалено сложна както е в някои операционни системи. Всъщност тази организация може да се използва и без TMOS за произволен gcc проект.

Всеки проект има една главна директория и произволен брой поддиректории, в които по някаква логика са систематизирани сорсовете. В главната директория се слага мейкфайла, който е универсален и може да ползва в различни проекти. За тоя мейкфайл има документация, ако някой се интересува, но по принцип това не е толкова важно, защото не би трябвало да се модифицира. Това, което се променя според конкретния проект са module.mk файловете. По един такъв трябва да има във всяка директория. Този файл е сравнително прост, защото обикновено в него само се изброяват сорс файловете и поддиректориите в текущата директория.

При компилация мейкфайла взема мк-файла от главната директория и ако в него има указани поддиректории се вземат и техните и така се получават всички сорсове. За да се добави нов сорс файл към проекта, просто трябва да се изброи в мк-файла на

директорията му.

Много често обаче се налага условна компилация. В TMOS има различни модули и библиотеки, но не е задължително те да се ползват от всяко приложение. За целта такива сорсове се изброяват в мк-файла чрез разни USE_XXX константи. След това ако константата се дефинира с 'y', всички сорсове дето зависят от нея ще участват в компилацията.

Освен да включват/изключват сорсовете може да се избират според стойността на определени параметри. Примерно ако някой стек има HAL ниво, то обикновено в поддиректории са различните портове като мк-файла тя се задава с параметър. Тоя тип параметри се кръщават като CFG_XXX.

Има различни начини на задаване стойности на параметрите, но основно се ползва директория boards. Там като поддиректории са конфигурации за различни проекти и в мк-файла са всички CFG и USE стойности, с излючение само на CFG_BOARD. Той не се задава чрез файл, а като параметър на мейк. Примерно:

make all CFG_BOARD=stm32-p103

Целта на цялата организация е да няма дублиране на код. В случая TMOS е нещо като набор от библиотеки. Колко от тях ще се ползват зависи от конкретното приложение, но то си има конфигуфация в boards и само там са неговите неща. За да се направи ново приложение, просто трябва да се добави нова конфигурация. Нищо друго не бива да се променя.

В конфигурационната директория освен параметрите в мк-файла може да има всякакви други специфични за приложението сорсове. Всъщност цялото приложение може да е тук. Но за предпочитане е тук да са само неща свързани с конфигурации, а приложението да си е отделен проект.

Приложенията могат да използват същата организация. Често се случва след първата ревизия на хардуера да има втора, трета... или пък да има различни опции на крайното изделие. Така приложението също може да има общи части, както и специфични конфигурации. Всяка конфигурация може да генерира различен фърмуер, но проекта си остава един.

Структура на директориите

Вече описах boards.

В директория hardware са хедърите на контролерите долу-горе така както би трябвало да ги дава производителя. За съжаление не ги дават в тоя вид... така че са преработвани. Систематизарани са по фамилии, архитектури и т.н. с цел да няма дублиране. Примерно често една и съща периферия се използва от много чипове, за целта се прави само една

дефиниция но на точното място, така че като се компилира даден проект, в него да влязат правилноте дефиниции.

В ports са портовете на кърнела на TMOS. В drivers са дравери естествено, а пък в services разни сървиси и стекове.

Кърнел

Кърнелът си е изчистен максимално, съвсем малък набор от функции и съвсем кратички... ппвечето на асемблер. Естествено нишки и т.н. няма да обяснявам, ще кажа само по-нестандартните неща.

Както казах няма флинтифлюшки като пускане спиране, разрешаване или други глупости. Кърнелът просто си тръгва и стартира main в нишка. Това е. Естествено потребителят ако иска други нишки ще трябва да си ги пусне. Но важното е, че мейна си е нишка, но е по-различна. Освен че си тръгва сама, мекн таска е също и idle task. С две думи в main не може да си викат блокиращи функции. Не бива и да се излиза от main. Иначе за другите нишки няма подобни изисквания.

Друга характерна особеност е, че всяка нишка си има 32 сигналчета по рождение. Сигнали се ползват драйверната система, така че ако потребителя иска да ползва сигналчета или не трябва да са от първите 8 или трябва да си ги резервира. Иначе когато се отваря хендъл към някой драйвер се заделя едно сигналче. После хендъла може да се ползва за блокиращи или неблокиращи заявки. За драйверите това няма никакво значение, те си получават заяката и като я обработят сигнализират клиента. На практика всички операции първо пращат заявката, а после си решават дали да блокират и за колко време да блокират...

Друга особеност касаеща нишките е, че кърнел функциите както и драйверните не използват стека на нишката. Така че няма нужда от презапасявания, пък и може да се пускат много нишко с малки стекове.

Драйверна система

Ос без драйвери е като предизборно обещание. От друга страна прекалено сложните драйвери също не са цвете. В случая с TMOS сме търсили компромис хем да е просто, хем универсално, хем да върши работа.

Простотата се постига с изчистения интерфейс - клиентът може да подава заяки за четене, писане и нещо като IO контроли. Универсалноссттта идва от това, че по отношение на четенето и писането за клиента няма никакво значение кой е драйвера. Примерно работата с UART, usb cdc че даже и tcp/ip сокет е по абсолютно един и същ начин. Разликата е само хендъл към какво се отваря.

Хендлът е обикновена структура, всъщност е POD C++ клас и има различни методи, които просто попълват данните заявката, информират системата за заявката и след това

евентуално чакат или не чакат за резултата. По-горе споменах, че при отваряне на хендъла от таск се заема и един от сигналите му. Така таска може да има няколко хендъла, да пуска заявки паралелно по тях и да реши кои резултати кога го интересуват. Въобще пълна свобода, което пък може да доведе и до объркване. За тая цел полето res винаги показва състоянието на хендъла и на заявката. Флагът FLG_CLOSED винаги показва дали хендъла е отворен, т.е. може да се ползва забзаявки към съответния драйвер. При пускане на заявка се вдига FLG_BUSY и докато е вдигнат клиента не бива да бара нищо, освен единствено да поиска канселиране, ако му е писнало да чака. Когато драйверът приключи със заяката сваля busy-то, сигнализира клиента и вдига FLG_SIGNALED, отделно може да вдигне и OK или ERROR. Дублирането на сигнала и флагче не е случайно, понякога клиента може да пусне няколко неща и да получи после няколко сигнала накуп. И ако един от тях е много важен, може да реши да се изнесе от функцията и да "забрави" че има и други необработени сигналчета. Така че флагчето си е нужно, но ако се ползват блокиращи методи на хендъла те си се грижат и за сигналчето и за флагчето.

От гледна точка на драйверите, те представляват една структура, в която има цялата информация за драйвера, затова се нарича info структура. Тук има една много голяма 'особеност', че всички вектори на прекъсвания сочат info структури, демек таблицата с векторите и таблицата с драйвери са едно и също нещо. Обикновено няма причина да се правят динамични драйвери, така че таблицата е статична, т.е. попълва се като константен масив at compile time. Не че не може и динамично, просто няма причини. Въпросът беше, че като се попълва таблицата с драйвери всеки трябва да си е на правилната позиция. Примерно драйверът за UART3 трябва да е на същия индекс като вектора за прекъсването му. Това изискване е при куртексите, при ARM7 позицията няма значение.

Инфо структурата има няколко задължителни полета в началото, сред тях са указатели към функциите за прекъсване (ISR), за контроли (DCR) и заявки (DSR). Това са функциите, които се викат от драйверната система или прекъсване. На тия функции винаги се подава като параметър указател към инфо структурата. Така драйверът си знае кой е, освен това в инфото е само хедър, след който всеки клас драйвери може да си има негови параметри и данни.

Обичайния ред е: първо при ресет системата вика всички DCR-и с DCR_RESET там драйверите могат да си направят нужната инициализация. След това като хукнат клиентите, почват да се пробват да отварят хендъли при което драйверът получава DCR_OPEN и кандидат хендъла. Тук драйверът може да одобри или откаже отварянето. Ако одобри, клиента може да почне да праща заявки, които се подават на DSR-а. Обикновено при получаването на първата заявка, драйверът пуска хардуера да чете, пише или каквото там се прави и излиза от DSR функцията. Когато приключи хардуера, обикновено прави прекъсване и се вика ISR-а, където драйвера сигнализира клиента на хендъла с ОК или грешка.

Междувременно може да е имало други заявки, но тъй като една периферия рядко може да прави няколко неща наведнъж, драйверът просто си нарежда чакащите хендъли в списък. И като приключи с един, хваща следващия и т. н. Казано простичко системата и драйверите се грижат за синхронизация между клиентите. Няма нужда един таск да мисли дали друг не ползва същата периферия. Даже драйвери като SPI поддържат заключване, т.е. ако клиента вдигне лок-а в една заявка драйверът се заключва към този клиент и не пуска други, включително чип селекта остава асертнат. Така клиента може да си нацепи операцията на няколко заявки. И няма нужда като оня бъглив код на Атмел от една друга тема да се копира памет, че да се получи една обща операция.

И инсталация...

Първо Yagarto или в краен случай друг тулчейн. После

svn checkout http://tmos.googlecode.com/svn/trunk/tmos

а за windows най-добре през еклипс, но първо трябва да се сложи svn клиент като Subclipse/Subersion, после вземате линка на репозиторито от страницата на проекта http://code.google.com/p/tmos/ и в перспективата SVN Repository го въвеждате като ново репозитори. Така може да се разглежда, в trunk/tmos е ос-а, а отделно има и нещо като примерчета в trunk/demoevals. В бъдеще може и други примерни проекти да направим. Та ако искате да си свалите някой проект - с десния checkout...

За съжаление файловете на Еклипс съдържат локални пътища и настройки, затова умишлено в не ги слагаме в репозиторитото (няма как да се синхронизират). Така след като се свали един проект, Еклипс не разпознава от какъв тип е. За целта трябва след сваляне да се пусне File->New->Convert to C/C++ project. След това може да се компилира в Еклипс, но ще изреве че няма CFG-TARGET. Най-добре в случая е с Manage Configurations да се направят конфигурации като се именоват така, както се казва съответната поддиректория в boards. Така след това в опциите C/C++ Build -> втория таб (Behaviour) като build команда се слага “all CFG_BOARD=${ConfigName}”, по същия начин и clean - “clean CFG_BOARD=${ConfigName}”. Това се прави за всички конфигурации. После само от чукчето се избира коя конфигурация да компилира.