Dokumentation zum over-the-air-update

Einführung

Das Rahmenprojekt des diesjährigen WSN-Praktikums war die Ausstattung des Solarhauses der TU Darmstadt mit einem Sensornetzwerk, welches verschiedene Sensordaten über einen Zeitraum von ca. 2 Wochen aufzeichnen sollte. Um dieses Netzwerk einfacher warten zu können, sollte es zudem mit einer over-the-air-Updatefunktion ausgestattet werden. Hierdurch wäre es möglich, remote eine neue Version der verwendeten Software in das Netzwerk einzuspielen. Obwohl die Softwareverteilungs- und –lademechanismen an sich funktionieren, konnte der OTA-Updater aufgrund der limitierten Hardware und des großen Umfangs der Aufzeichnungssoftware für die Messdaten nicht im Solarhaus eingesetzt werden.

Im Folgenden bezeichnen wir das Modul, das über den Bootstraploader auf den Mote installiert wird (mittels **make <project>.upload CORE=<project>.z1**), als Firmware, und das Modul, welches dynamisch über den OTA-Updater verteilt wird, als die ELF-Datei.

Die Firmware lässt sich grob in zwei Teile gliedern: Den Mechanismus zur Verteilung der ELF-Datei über Funk, sowie den Lademechanismus für die ELF-Datei.

Der Update-Prozess

Wahl des Protokolls

Bei Contiki werden verschiedene Protokolle mitgeliefert, die prinzipiell für die Aufgabe, große Datenmengen zu verteilen, in Frage kommen, allen voran das Deluge-Protokoll, welches theoretisch zum einen die Verteilung der Daten managen soll, und sich zum anderen auch um die Aktualität der Daten kümmert. In der Praxis versagte es jedoch in beiden Aufgaben. Da der Code zudem schwer verständlich und schlecht kommentiert ist, mussten wir weitere Optionen evaluieren.

Bei Contiki werden noch ein paar andere Protokolle mitgeliefert (Rudolph, Trickle), die in die engere Auswahl kamen. Jedoch erfüllte keines die Anforderungen komplett, weshalb wir das MOAP-Protokoll implementiert haben. Eine Beschreibung dieses Protokolls findet sich in Foliensatz 8 (http://www.dvs.tu-darmstadt.de/teaching/wsnlab/2011/slides/08.mw.pdf) auf Seite 21. Das Protokoll wurde leicht abgewandelt, damit jeder Knoten als Quelle dienen kann und die jeweils aktuellste Version verteilt wird. Deshalb enthält die Publish-Nachricht immer die Versionsnummer, die der Knoten momentan verteilt, und jeder Knoten sendet periodisch eine Publish-Nachricht, wobei sich das Intervall immer weiter vergrößert, wenn keine Subscriptions eingehen.

Probleme bei der Implementierung

Bei der Implementierung des Protokolls stießen wir auf einen Bug im Compiler. Wenn man über Broadcast eine struct versendet, die einen oder mehrere **uint16_t** enthält, ist beim Empfang die Endianness vertauscht. Diese muss man dann manuell wieder tauschen. Erstaunlicherweise tritt dieser Fehler nicht bei der Simulation mit Cooja auf.

Bei der Simulation des OTA-Updaters mit Cooja muss man daher in der Datei **distrib.h** die Zeilen

//#define COMPILE_COOJA
#define COMPILE_REAL

in

#define COMPILE_COOJA
//#define COMPILE_REAL

ändern. Dadurch wird das manuelle Tauschen der Endianness in der Firmware deaktiviert.

Der ELF-Loader

Grundlegendes

Die im Netzwerk verwendete Firmware wird als ELF-Datei (Executable and Linkable Format) verteilt. Die Datei wird vom Contiki ELF-Loader in den Flash-Speicher geladen. Contiki unterstützt die Erzeugung von ELF-Dateien, indem man das Kommando **make filename.ce** eingibt. Hierbei wird die Datei **firmware.c** in eine ELF-Datei umgewandelt. Die in der Datei referenzierten Funktionen werden jedoch nicht automatisch mit der ELF-Datei gelinkt. Es ist daher nötig, manuell die fehlenden Objektdateien zu linken, die den Code enthalten, der noch nicht in der ELF-Datei enthalten ist.

Man kann eine ELF-Datei entweder ins RAM, oder in den Flash-Speicher laden. Dies wird über die Variable **ELFLOADER_CONF_TEXT_IN_ROM gesteuert**. Das Speichern im RAM hat zwei Nachteile. Zum einen ist die Größe der ELF-Datei hierbei stark begrenzt, zum anderen ist die Speicherung nicht persistent, d.h., nach jedem Neustart muss die ELF-Datei erneut geladen werden, was jeweils bis zu einigen Minuten dauern kann. Daher unterstützt der OTA-Updater ausschließlich die Speicherung im ROM.

Allgemeines zum Contiki ELF-Loader

Der ELF-Loader von Contiki besteht aus einem generischen sowie einem hardwareabhängigen Teil. Der generische Teil ist hauptsächlich unter **core/loader/elfloader.c** untergebracht, der hardwarespezifische für die Z1- und Skymotes in **core/loader/elfloader-msp430.c**. Die Maximalgrößen des Text- und Datensegments der ELF-Dateien lassen sich im Makefile über die Variablen **ELFLOADER_DATAMEMORY_SIZE** bzw. **ELFLOADER_DATAMEMORY_SIZE** steuern. Nach einer Änderung muss unbedingt **make clean** gemacht werden.

Die maximale Größe einer ELF-Datei ist bei den Z1-Motes stark limitiert. Theoretisch nutzbar sind maximal 9000 Byte, jedoch gehen durch Alignment 352 + 458 = 810 Byte verloren, weshalb lediglich 8190 Byte in der Praxis verfügbar sind. Das Alignment ist erforderlich, da beim Schreiben in den Flash-Speicher jeweils Seiten von 512 Byte gelöscht und beschrieben werden.

Anpassungen am Contiki ELF-Loader

Wir haben einige Anpassungen am Contiki ELF-Loader vorgenommen, um unsere ELF-Datei laden zu können. Zum einen haben wir die maximale Länge für Symbole (Symbole sind z.B. Funktions- oder Variablennamen) von einem konstanten Wert in eine Variable namens **ELFLOADER_MAX_SYMBOLSIZE** geändert, die im Makefile angegeben werden kann.

Desweiteren haben wir sichergestellt, dass wenn **ELFLOADER_CONF_TEXT_IN_ROM** gesetzt ist, die Speicherung auch wirklich im Flash-Speicher erfolgt. Daher haben wir die Deklaration des

Speicherbereichs für die ELF-Datei entsprechend angepasst, damit der Code der ELF-Datei immer im Flash gespeichert wird.

Compilierung eines ELF-Moduls

Um ein ELF-Modul zu compilieren, stehen 2 PHP-Skripte zur Verfügung, die den Vorgang vereinfachen. Zum einen gibt es das Skript **checksymbols.php**, welches prüft, ob die ELF-Datei von der Firmware geladen werden kann. Als ersten Parameter erwartet es den Pfad zum Firmware-Modul (die .z1-Datei), als zweiten Parameter den Pfad zum ELF-Modul. Es gibt alle im ELF-Modul fehlenden Symbole aus.

Die Erstellung eines ELF-Moduls wird durch das Skript **linkelf.php** stark vereinfacht. Als ersten Parameter erwartet es den Pfad zur Firmware, als zweiten den Pfad zu dem Ordner, in dem die Objektdateien, die zu der ELF-Datei gehören, gespeichert sind. Dies ist normalerweise der Ordner **obj_z1**, der „/“ am Ende der Pfadangabe muss unbedingt vorhanden sein. Als dritten Parameter erwartet das Skript den Namen der ELF-Datei. Die ELF-Datei muss vorher unter obj_z1 kopiert werden, nachdem sie mit **make firmware.ce TARGET=sky** erstellt wurde. Vorher sollte man jedoch ganz normal das zum ELF-Modul gehörende Projekt compilieren, um die Objektdateien zu erzeugen. Das PHP-Skript sucht automatisch nach fehlenden Symbolen und versucht, diese mit der ELF-Datei zu linken. Dies geht in vielen Fällen gut, jedoch leider nicht in allen.

Zum einen kann es vorkommen, dass Objektdateien in Unterverzeichnissen gespeichert sind. Sollten sich in **obj_z1** Unterverzeichnisse befinden, so muss man die Objektdateien von dort in **obj_z1** kopieren.

Zudem gibt es ein Problem mit den Funktionen **memset** und **memcpy**. Es empfiehlt sich, die Datei **workaround.o** aus dem SVN in den Ordner **obj_z1** zu kopieren, um derartige Probleme zu vermeiden. Der dadurch behobene Fehler beruht auf einer Art Workaround von Contiki, der dafür sorgt, dass **memcpy** und **memset** als **w_memcpy** respektive **w_memset** aufgerufen werden, weshalb man hier beim Linken der ELF-Datei Probleme mit fehlenden Symbolen bekommt.

Als letztes gibt es noch einige Funktionen, die generell nicht unter **obj_z1** auftauchen, da sie aus der libgcc gelinkt werden. Wenn dieser Fehler auftritt, sollte man am besten in der Datei **contiki-z1.map** nach dem Namen der fehlenden Funktion suchen. Dann findet man eine **.a**-Datei, die man nach **obj_z1** kopiert und mittels **msp430-ar –x <filename>** dorthin entpackt.

Upload eines ELF-Moduls

Nachdem das ELF-Modul compiliert wurde, kann man es auf einen Mote hochladen. Nach dem Linken befindet sich in **obj_z1** eine Datei **output**. Diese kopiert man in das Verzeichnis, wo der Uploader ist. Nun führt man folgende Befehle aus, um den Upload durchzuführen:

base64 –w0 output > output.encoded

./uploader | serialdump –b115200 <motedevice>

Danach loggt man sich mit **serialdump –b115200 <motedevice>** auf die Shell ein und gibt folgende Befehle ein:

<return>

decode

update

Verbesserungspotenzial

Es gäbe einige Möglichkeiten, den OTA-Updater zu optimieren. Aus Zeitmangel wurden diese zwar nicht umgesetzt, es soll aber im Folgenden kurz darauf eingegangen werden.

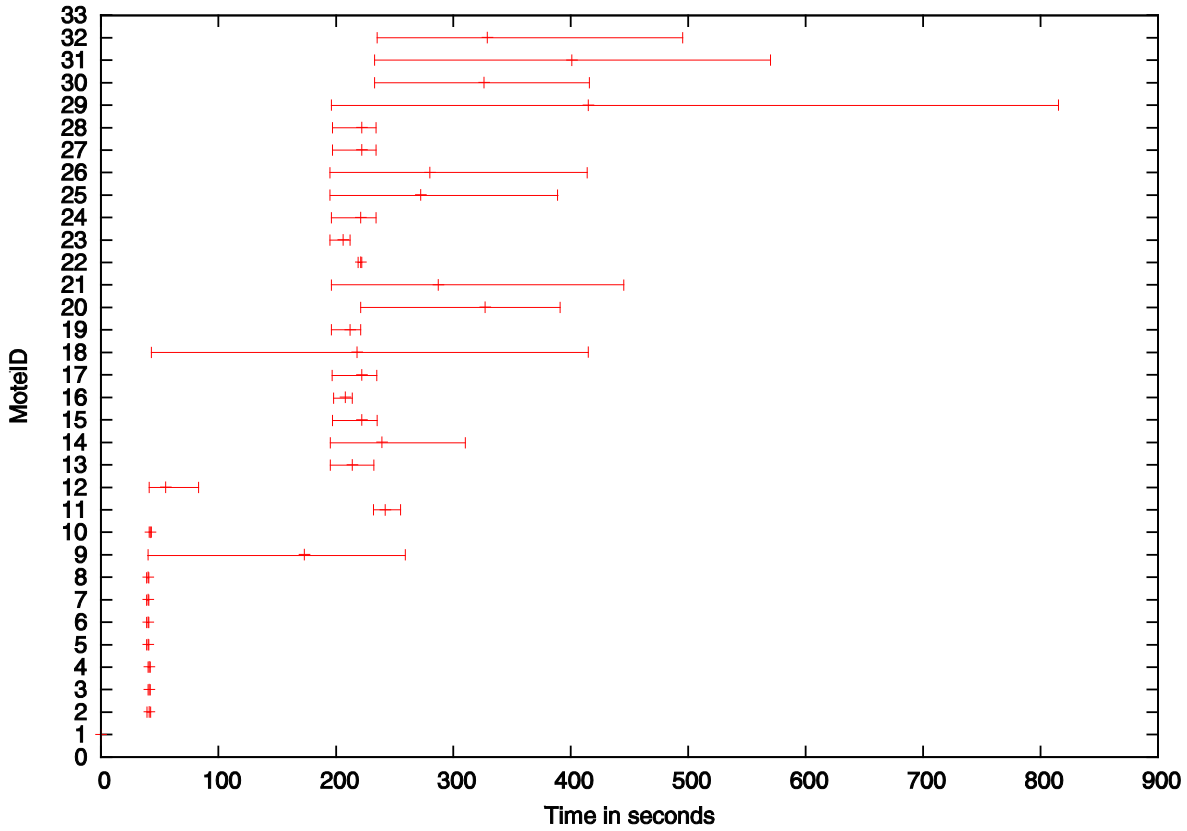
Zum einen haben die Z1-Motes ein ganzes 64k-Segment, was derzeit komplett ungenutzt ist, da der aktuelle Compiler es nicht unterstützt, dort Programmcode abzulegen. Es dürfte zwar möglich sein, manuell Code dort abzulegen, jedoch verwendet der Compiler standardmäßig nur 16-Bit-Pointer und normale Returns. Man müsste dementsprechend darauf achten, einen Long Return zu machen und längere Pointer zu benutzen. Dann wäre es wahrscheinlich möglich, den Code für den Updater und den Loader dort unterzubringen, wodurch man volle 64k für die zu verteilende ELF-Datei übrig hätte.

Man müsste dann auch nicht mehr auf das ELF-Format setzen, sondern könnte auch komplette Images verteilen. Das hätte den Vorteil, dass man sich die mühsame Compilierung der ELF-Datei spart, jedoch als Nachteil mehr Daten zu übertragen hätte.

Evalutation der Performance des Update-Prozesses

Zum Test der Performance des OTA-Updaters wurde eine 1120 Byte große ELF-Datei im Piloty-Testbed verteilt. Eine Beschreibung der Testumgebung kann man unter http://www.dvs.tu-darmstadt.de/research/wsn/ abrufen. Der Updateprozess wurde von Knoten 1 aus gestartet, und es wurde jeweils gewartet, bis die ELF-Datei im kompletten Netzwerk verteilt war.

Es wurden insgesamt 3 Testläufe durchgeführt. In der folgenden Grafik sieht man jeweils zu jedem Knoten die minimal, durchschnittlich und maximal benötigte Zeit für ein Update für jeden Knoten.



Die maximal benötigte Zeit für ein komplettes Update waren hier ca. 800 Sekunden. Verglichen mit der Zeit, die man benötigt, um 32 Knoten manuell zu programmieren, hat man einen starken Zeitgewinn durch die Verwendung des OTA-Updaters. Bei Programmen, die die Größenlimits nicht überschreiten, ist es daher empfehlenswert, auf den OTA-Updater zur Verwaltung der Knoten zurückzugreifen.