Gömülü Sistem Yazılımlarının Geliştirilmesinde Aygıt Ağacı Yapısının Kullanılmasına Yönelik bir Çalışma

A Study on the Use of Device Tree Structures for Embedded Software Development

Sadık Arslan, Ercüment Türk
Kent Kart Ar-Ge Merkezi
Kent Kart Ege Elektronik A.Ş.
İzmir, Türkiye
{sadik.arslan, ercument.turk}@kentkart.com.tr

Geylani Kardaş
Uluslararası Bilgisayar Enstitüsü
Ege Üniversitesi
İzmir, Türkiye
geylani.kardas@ege.edu.tr

***Özetçe*—Mikro işlemci içeren gömülü sistemlerde işletim sistemi çekirdeklerine saat hızı, bacak ismi, aygıt adresi gibi birçok donanımsal bilgi girilmesi gerekmektedir. Klasik işletim sistemi çekirdeği derlenmesi işleminde, gömülü sistemin desteklemiş olduğu bütün özelliklerin bilgileri kaynak kodların içerisinde bulunmaktadır. Bu bilgiler üzerinde yazılım geliştirme çalışmaları yapılırken işletim sisteminin çekirdeğinin de her denemede tekrar derlenmesi gerekmektedir. Yüzlerce farklı özelliğin bulunduğu sistemlerde bu süreç oldukça zaman alıcı ve maliyetli olmaktadır. Öte yandan Aygıt Ağacı veri yapısı kullanılarak gömülü sistemlerde fiziksel bileşenlerin tanımlanması yapılabilmektedir. Bu yapı ile işletim sisteminin çekirdeği bir kere derlenmekte ve gerekli geliştirme çalışmaları bu yapı üzerinden yürütülebilmektedir. Bu çalışmada, Aygıt Ağacı yapısının gömülü sistem yazılımlarının geliştirilmesine yönelik uygulanabilecek bir yöntem tanıtılmaktadır. Klasik yöntem ile Aygıt Ağacı yapısının kullanımını içeren yeni yöntem karşılaştırılmıştır ve yeni yöntemin uygulanmasının güçlü ve zayıf yönleri tartışılmıştır.**

Anahtar Sözcükler—Aygıt Ağacı yapısı; gömülü yazılım; mikro işlemciler; işletim sistemleri

***Abstract*—Embedded systems with microprocessors require many hardware information such as clock speed, port name, and device address to be entered into the operating system cores. In the classic compilation process for operating system kernel, all features, that the embedded system supports, need to be included in the source code. The kernel of the operating system must be recompiled for each software development process which considers related information about these features. In systems that have hundreds of different features, this process is time consuming and causes high cost as expected. Use of Device Tree data structure for the specification of physical components in embedded systems may facilitate this software development process since the kernel of the operating system is compiled just once in case of applying a development process based on this data structure. In this study, a methodology for the development of embedded software with using the Device Tree structure is introduced. New methodology is compared with existing software development approach and a preliminary evaluation is performed with taking into account pros and cons of applying this new methodology.**

Keywords—Device Tree structure; embedded software; micro processors; operating systems

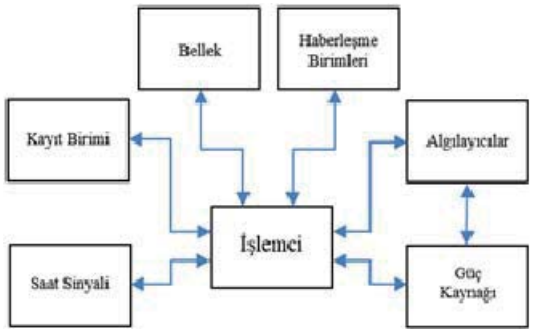
I. GİRİŞ

Gömülü sistemler, kişisel bilgisayarlardan farklı olarak, kendisi için önceden özel olarak tanımlanmış görevleri yerine getiren sistemlerdir [1]. Gömülü sistemlerde kısıtlı güçte bir işlemci, bir bellek ve diğer yardımcı birimler kullanılır. Gömülü Linux, Android, JavaOS, LynxOS, Mobilinux, Windows CE gibi birçok işletim sistemi de gömülü sistemlerde kullanılmaktadır. Cep telefonları, araç takip cihazları, ağ donanımları, motor denetleyiciler, fren sistemleri, ev otomasyon ürünleri, hava savunma sistemleri, tıbbi ekipmanlar, ölçüm sistemleri gömülü sistemlere örnek olarak verilebilir.

Genel olarak bir gömülü sistemin yapısında işlemci, güç kısmı, bellek, kayıt birimleri, haberleşme ara yüzleri, saat sinyalleri, algılayıcılar gibi bölümler bulunmaktadır [1]. Şekil 1'de basitleştirilmiş bir gömülü sistem blok çizimi görülebilir.

Günümüzde sürekli değişen ve gelişen gömülü sistemlerde çok farklı donanımsal çevreseller kullanılmaktadır. Bununla birlikte işletim sistemleri ve işletim sistemi çekirdekleri (ing. kernel) de oldukça sık bir şekilde değişmektedir. Birçok farklı kuruluş ve firma farklı Yonga Üzerinde Sistem (ing. System on a Chip) (SOC) platformları için sürekli işletim sistemi dağıtımları yayınlamaktadırlar.

978-1-5386-0930-9/17/$31.00 ©2017 IEEE



Şekil 1. Basitleştirilmiş bir gömülü sistem blok çizimi [2].

Aygıt Ağacı (ing. Device Tree) (DT) bir gömülü sistem donanımının içerisindeki fiziksel cihaz bileşenlerinin düğümler ile tanımlanmasını sağlayan bir veri yapısıdır [3, 4]. "Open Firmware", "Power Architecture Platform Requirements" gibi standartlar kapsamında da kullanılan DT, bir gömülü sistem içerisindeki donanım bileşenlerinin eksiksiz bir teknik tanımının yapılmasını sağlamaktadır [5]. İşletim sistemi çekirdekleri (ing. kernel) bir kere derlendikten sonra farklı donanım konfigürasyonları düzenlenmiş farklı ağaç yapıları ile daha geniş bir mimari işlemci ailesi içerisinde desteklenebilmektedir. DT, Linux işletim sistemi çekirdeği ARM, x86, MicroBlaze, PowerPC ve SPARC işlemci mimarileri ile çalışabilmektedir. Özellikle ARM işlemci ailesinde çok fazla firma çok farklı sayıda mikro işlemci ürünlerini piyasaya sürmektedirler. Bu nedenle DT kaynak dosyası kullanımları ARM platformları için oldukça önemlidir [5].

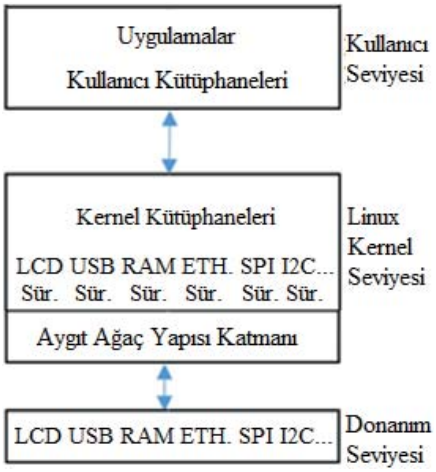
DT dosyası desteği bulunmayan işletim sistemi çekirdeklerinde tüm çevresel birimlerin sürücü dosyaları çekirdek kaynak kodunun içinde bulunmaktadır. Çekirdeklerde saat sinyali frekansı, bacak ismi, kesme bacağı gibi donanımsal bilgileri değiştirebilmek için çekirdekteki kaynak kodunun değiştirilmesi gerekmektedir. Bu işlem de hem bakım maliyeti hem de zaman açısından problemli olmaktadır. DT yaklaşımı ile bu problemlerin önüne geçilmeye çalışılmaktadır. İşletim sistemi çekirdeği içerisinde bulunan donanım özellikleri, derlenen bir aygıt ağacı yapısına dönüştürülmekte ve çekirdekten dışarıya alınmaktadır. Çekirdek derlenirken donanımsal sistemin özellikleri dışlanmış olmakta ve genel kullanım amaçlı bir işletim sistemi çekirdeği elde edilmektedir.

Günümüze kadar yapılan çalışmalar incelendiğinde gömülü sistemlere ait yazılımların geliştirilmesinde DT kullanımını içeren oldukça az sayıda çalışma olduğu görülmektedir [6 - 10]. Ülkemizde DT'lerin gömülü sistem uygulamalarının geliştirilmesi amacıyla kullanımına yönelik bir çalışmaya da rastlanmamıştır. DT yazılımı geliştirme çalışmaları göz önüne alındığında Likely ve Boyer [6], DT yazılımlarındaki düğüm tipleri, hiyerarşisi, söz dizimi konularını ele almış ve bir platformda DT'nin kullanımının deneysel bir çalışmasını gerçekleştirmiştir. Bunun yanı sıra sanal makine ve PCI ara yüzlerinin oluşturulmasında DT kullanımını anlatan çalışmalar bulunmaktadır (örneğin [7] ve [8]). Nicolescu ve Mosterman, [9]'da FPGA tasarımında DT kullanımından söz etmektedir. Jassi ve ark. [10], gömülü sistemler için IEEE 1685-2014 IPXACT [11] standardı kullanılarak hazırlanan ve donanım sürücü kodu üreten bir sistem önermiştir. Bu çalışmada tek bir platform için kameradan video çalıştırma uygulaması göz önüne alınmıştır. Çalışmada DT üretimi yapıldığı belirtilmiştir. Ancak tüm bu çalışmalarda DT'nin tasarımı ve yazılım geliştirmede kullanılmasına yönelik ayrıntılar bulunmamaktadır ve araştırmaların deneysel sonuçları nadiren yer almaktadır. Bu noktadan hareketle bu çalışmada DT yapısının bir gömülü sistem yazılımının geliştirilmesine yönelik uygulanabilecek bir yöntem tanıtılmakta ve klasik yazılım geliştirme yöntemine göre DT tabanlı yeni yöntemin avantaj ve dezavantajları anlatılmaktadır.

Bu bildirinin 2. bölümünde çalışmada kullanılan ortam ve Linux işletim sistemi tanıtılmaktadır. Bölüm 3'te çalışmada DT yapısının bir gömülü sistemin üzerinde nasıl uygulandığı anlatılmaktadır. Bölüm 4'te geliştirilen sistemin uygulama sonuçları ve değerlendirmesi bulunmaktadır. Son bölümde ise çalışma sonuçları ve gelecekte yapılabilecek çalışmalar verilmiştir.

II. SİSTEMİN YAPISI VE ÇALIŞMASI

Geliştirilen mikro işlemci tabanlı gömülü sistem, ilk aşamada toplu taşıma araçlarında sürücü bilgisayarı olarak kullanılacaktır. Toplu taşıma araçlarında radyo frekansına dayalı (ing. Radio frequency) (RF) kart okuma, yolcu bilgilendirme, ekranlar ve hoparlörler kullanılarak araç içi reklam oynatma gibi görevleri bu cihaz yerine getirecektir. Üzerinde çalışılan bu gömülü sistemde Linux Ubuntu işletim sistemi kullanılmıştır. Mikro işlemci olarak da NXP firmasının iMX6 serisinden [12] bir çift ARM çekirdekli bütünleşmiş devre kullanılmıştır. Sistemde bulunan kullanıcı uygulamaları, temelde kullanıcı kütüphaneleri ve çekirdek kütüphanelerine erişerek, tanımlanmış olan çalışmalarını (ing. execution) gerçekleştirmektedir. Çekirdek de donanım yönetimi için kullanılan Evrensel Seri Veriyolu (ing. Universal Serial Bus) (USB), Seri Çevresel Veriyolu (ing. Serial Peripheral Interface Bus) (SPI), Bütünleşmiş Devreler Arası Hat (ing. Inter-Integrated Circuit) (I2C) gibi birçok farklı ara yüz ve özellik için bulunan sürücüleri kullanmaktadır. Donanımların özellikleri çekirdek sürücülerine DT dosyaları aracılığı ile iletilmektedir. Bu çalışmada kullanılan gömülü sistemin basit yapısı Şekil 2'de görülebilmektedir.



Şekil 2. Çalışma yapılan sistemin katmanlı yapısı.

ARM mikro işlemcisi bulunan bu yapıda, Linux Kernel tabanlı işletim sistemleri, güç aldığında, çekirdek dosya sisteminde ilk olarak *arch/arm* altında bulunan kendi sistemine uygun dosyadan çalışmaya başlar. Çekirdek başlangıçta işlemci ve bellek ayarlamalarını yapar ve sistemin DT yapısını destekleyip desteklemediğini belirler. DT kullanımı çekirdek açılış zamanlamasını çok fazla değiştirmemektedir. Platform kodu sürücüleri aygıt modeline kaydetmekte, sonrasında da aygıt sürücülerini bunlara bağlamaktadır. USB gibi bazı aygıtlar için DT ihtiyacı olmayabilir ancak bu durum mikro işlemci donanımı ile ilgilidir. DT yapısının asıl etkisi sistemin açılış sıralaması üzerinde değil; çekirdeğin sisteme bağlı çevre birimleri hakkında bilgi aldığı alanlar üzerindedir.

DT kullanan gömülü platformların çoğu, *of_platform* veri yolu altyapısından yararlanmaktadırlar. Platform veri yolu gibi *of_platform* veri yolu da aslında bir donanımsal veri yolunu ifade etmez. Bu yapılar, donanımsal aygıtları aygıt modeline manüel olarak kaydetmek için bir yazılım kurgusudur. Bu yapı işletim sistemi çekirdeğine bağlanmaya çalışan donanımlar için oldukça kullanışlıdır. Donanımların sistem çalışırken eklenebilmesini ve çıkarılabilmesini sağlamaktadır.

platform ve *of_platform* veri yollarının çekirdekleri neredeyse aynıdır. Ancak bu veri yolları aygıtları sürücüler ile eşleştirmede farklı yöntemler kullanmaktadırlar. *platform* veri yolu aygıt ile sürücüyü *.name* niteliklerini (ing. property) ortak olarak taşıyor ise eşleştirmektedir. *of_platform* veri yolu ise DT'den alınan değerlere dayanarak sürücüleri cihazlara eşleştirir. Bu işlemde çekirdek sistemi özellikle, *name, device_type* ve *compatible* niteliklerini eşleşme tablosuna göre dikkate almaktadır. Şekil 3'te örnek olarak bir Gerçek Zaman Saati (ing. Real Time Clock) (RTC) I2C aygıtında DT ve sürücü dosyalarındaki yapılar görülebilmektedir. Sürücü kodu o*f_match* tablosundaki *.compatible* eşleşmelerinden bir tanesinin tutması gerekmektedir.

```
//Epson RX8900 DT yapısı
rtc: rtc@32 {
    compatible = "epson,rx8900"
    reg = <0x32>;
    epson,vdet-disable;
    trickle-diode-disable;
};
//Epson RX8900 sürücü eşleşme tablosu kodu
static const struct of_device_id rv8803_of_match[] = {
{
    .compatible = "microcrystal,rv8803",
    .data = (void *)rx_8900
},
{
    .compatible = "epson,rx8900",   //eşleşme olur
    .data = (void *)rx_8900
},
{ }
};
```

Şekil 3. Bir RTC I2C aygıtına ait örnek dosya yapısı

Buraya kadar anlatılan aygıt bağlama yöntemlerinden olan *of_platform* mecburi bir yöntem değildir. Klasik yöntem olan *platform* veri yolu yöntemi zaten tüm sürücülerde bulunmaktadır. Bazı sürücü geliştiricileri *of_platform* veri yoluna geçiş yapmamışlardır. *of_platform* veri yolu desteği bulunan işletim sistemi çekirdeklerinde *platform* veri yolu desteği aynı anda bulunmaktadır ve her iki yöntem de aynı anda çalışabilmektedir [6].

III. AYGIT AĞACI ÇALIŞMASI

DT en basit ifadeyle, donanım yapılandırmasını açıklayan bir veri yapısıdır. Bu yapı, gömülü sistemin işlemcisi, belleği, veri yolları ve çevre birimleri hakkında bilgileri içerir. İşletim sistemi, önyükleme sırasında DT yapısını ayrıştırır ve mikro işlemciyi nasıl yapılandıracağı ve hangi aygıt sürücülerinin yükleneceği konusunda kararlar almak için bu yapıyı kullanır.

DT yapısı, "/" simgesi ile temsil edilen, kök olarak adlandırılan bir düğümle gösterilmektedir. Yapıda her düğümden çok sayıda, ismi olan ve bir numarası olabilen, çocuk düğüm çıkabilmektedir. Düğümler isteğe bağlı olarak, keyfi ek veri içeren nitelik değerlerine de sahip olabilirler. DT yapısında bulunan verilerin biçimi, IEEE 1275-1994 standardı tarafından daha önceden belirlenmiş olan kurallara çok yakındır [13].

Aygıt Ağacı Kaynağı (ing. Device Tree Source) (.dts) dosya biçimi, aygıt ağaçlarını yazılım geliştiriciler tarafından düzenlenebilir biçimde ifade etmek için kullanılır. Aygıt Ağacı Derleyici Aracı (ing. Device Tree Compiler Tool) (dtc), DT'nin .dts biçimini işletim sistemleri tarafından ihtiyaç duyulan İkili Aygıt Ağacı (Binary Device Tree Blob) (dtb) biçimine çevirmek için kullanılmaktadır. Şekil 4'te .dts yapısının kök içeren basit bir hali verilmiştir. Görüldüğü üzere düğümlerin içersinde birçok farklı özellikte bilgi girilebilmektedir.

```
/ {                             // kök düğümü
    bir-nitelik;                //bir donanım özelliği
    bir-cocuk-dugum {           //ilk çocuk düğüm
        dizi-niteligi = <0x10 53>;
        yazi-niteligi = "merhaba, dunya";
    };
diger-cocuk-dugum {             //diğer çocuk düğüm
        binary-nitelik = [0221ALI];
        yazi-listesi = "evet","hayir","belki";
    };
};
```

Şekil 4. Kök içeren basit .dts yapısı

DT yapısı bir mikro işlemci sistemi için tam bir donanımsal aygıt listesi değildir. USB gibi bazı özellikler doğrudan ayarlanmış olabilmektedir. DT yapısında olmayan özellikler de işletim sistemi tarafından çalıştırılmaktadır. Bu durumun ayarlanması mikro işlemciye özel işletim sistemi kodu geliştiricileri tarafından yapılmaktadır.

Gerçekleştirilen çalışmada öncelikle DT yapısının kök düğümü tasarlanmıştır. Şekil 5'te NXP firmasına ait i.MX6 DualLite mikro işlemcisi kullanılarak bu çalışma için tasarladığımız bir gömülü sistem kök düğümü yapısı görülebilir. DT'nin başında bu kök düğümü yer almaktadır. Burada *model* ve *compatible* nitelikleri tasarlanan sistemin tam adını <mfg>,<board> şeklinde oluşturmaktadır. <mfg> sistemin üreticisini, <board> ise model numarasını

içermektedir. Bu yapı tasarlanan kart için tek tanımlayıcı olmaktadır. *Compatible* niteliğinin kullanımı zorunlu değildir. Ancak, aynı donanıma sahip farklı sistemlerin uyumluluğunun tanımlanması için kullanılmaktadır.

```
/ {
    model = "Kentkart, i.MX6 DualLite Smarc Board";
    compatible = "kk,imx6dl-smarc", "kk,imx6dl";
};
```

Şekil 5. Tasarlanan sistemin kök düğümü

Sisteme ait her işlemci için bir düğüm yer almaktadır. İşlemcilerin tanımlanması için kök düğümün bir alt düğümü olan *cpus* düğümü bulunmaktadır. Her işlemci de *cpus* düğümümün alt düğümü olarak yerleşmektedir. İşlemci düğümlerin tanımında kesin olarak bulunması gereken bir özellik yoktur. Ancak *#address-cells=<1>* ve *#size-cells=<0>* gibi bazı nitelikleri vermek uygulama açısından faydalı olmaktadır. Bunlar düğümlerde bulunan *reg* niteliğinin boyutlarını ve yapısını tanımlamaktadır. Kullanılan *reg* niteliği ise işlemcilerin fiziksel numaralarını kodlamak için kullanılmaktadır. İşlemci düğümleri için birim adı *cpu@0* biçimindedir. Bunlar ile birlikte Şekil 6'daki örnekte verildiği gibi düğümlerde sistemin ihtiyacına göre farklı özellikler de tanımlanabilmektedir. Örneğin, *next-level-cache* ile işlemci önbelleği bilgileri, *operating-points* ile işlemcinin çalışacağı frekanslardaki besleme voltajları, *clocks* ile işlemci içerisindeki saat sinyallerinin hangi saat kaynaklarından bağlanacağı, *arm-supply* ile işlemcinin ARM çekirdeğinin hangi kaynaktan besleneceği gibi bilgiler de burada gösterilmiştir.

```
/ {
    cpus {
    #address-cells = <1>;
    #size-cells = <0>;
        cpu0: cpu@0 {
        compatible = "arm,cortex-a9";
        device_type = "cpu";
        reg = <0>;
        next-level-cache = <&L2>;
        operating-points = <
                /* kHz    uV */
                996000  1275000
                792000  1175000
                396000  1150000
        >;
        fsl,soc-operating-points = <
                /* ARM kHz  SOC-PU uV */
                996000  1175000
                792000  1175000
                396000  1175000
        >;
        clock-latency = <61036>;
        clocks = <&clks IMX6QDL_CLK_ARM>,
        <&clks IMX6QDL_CLK_PLL2_PFD2_396M>,
        <&clks IMX6QDL_CLK_STEP>,
        <&clks IMX6QDL_CLK_PLL1_SW>,
        <&clks IMX6QDL_CLK_PLL1_SYS>,
        <&clks IMX6QDL_PLL1_BYPASS>,
        <&clks IMX6QDL_CLK_PLL1>,
        <&clks IMX6QDL_PLL1_BYPASS_SRC> ;
        clock-names = "arm", "pll2_pfd2_396m", "step",
                "pll1_sw", "pll1_sys", "pll1_bypass", "pll1",
                "pll1_bypass_src";
        arm-supply = <&reg_arm>;
        pu-supply = <&reg_pu>;
        soc-supply = <&reg_soc>;
    };
    cpu@1 {
        compatible = "arm,cortex-a9";
        device_type = "cpu";
        reg = <1>;
        next-level-cache = <&L2>;
    };
};
```

Şekil 6. Tasarlanan sistemin işlemci düğümü

Sistemin ana belleği gibi kök düğümün bir çocuğu olan *memory* düğümü yazılmaktadır. Burada *reg* niteliği, kullanılabilir hafızadaki bir veya daha fazla fiziksel adres aralığını tanımlamak için kullanılır. Şekil 7'de tasarlanan sistemde kullanılan *memory* düğümü görülebilir.

```
memory {
    reg = <0x10000000 0x40000000>;
};
```

Şekil 7. Tasarlanan sistemin bellek düğümü

DT yapısında, sistemdeki veri yollarını ve aygıtları tanımlamak için bir düğüm hiyerarşisi kullanılır. Her veri yolunun veya aygıtın ağaç yapısında kendi düğümü bulunmaktadır. Aygıtların ağaç yapısındaki durumları sistem tasarımları değiştikçe oldukça sık değişmektedir. Geliştirme esnasında en çok müdahale edilen kısım bu kısımdır. Şekil 8'de bizim uygulamamızdan verilen örnekte kök düğümün altında bulunan *regulators, sound* ve *lcd@0* düğümleri görülebilmektedir. Sistemdeki voltaj düzenleyiciler olan *regulators* düğümünün altında farklı voltaj çıkışlarının verilmesini sağlayan, *reg_1p8v, reg_2p5v* ve *reg_3p3v* olarak isimlendirilmiş 3 adet düğüm bulunmaktadır. Sistemin ana ses çıkışını sağlayan aygıt *sound* düğümü ile verilmiştir. Bu düğümde ses işlemlerini yapan dönüştürücü entegre devrenin tanımı ve bazı özellikleri nitelikler şeklinde verilmiştir. Bunlarla birlikte örnek olarak cihazda görüntü çıkışını sunan ekranın özelliklerinin verilmesini sağlayan düğüm *lcd@0* olarak görülebilmektedir.

```
regulators {
    compatible = "simple-bus";
    #address-cells = <1>;
    #size-cells = <0>;
    reg_1p8v: 1p8v {
        compatible = "regulator-fixed";
        regulator-name = "1P8V";
        regulator-min-microvolt = <1800000>;
        regulator-max-microvolt = <1800000>;
        regulator-always-on;
    };
    reg_2p5v: 2p5v {
        compatible = "regulator-fixed";
        regulator-name = "2P5V";
        regulator-min-microvolt = <2500000>;
```

```
            regulator-max-microvolt = <2500000>;
            regulator-always-on;
        };
        reg_3p3v: 3p3v {
            compatible = "regulator-fixed";
            regulator-name = "3P3V";
            regulator-min-microvolt = <3300000>;
            regulator-max-microvolt = <3300000>;
            regulator-always-on;
        };
};
sound {
    compatible = "fsl,imx-audio-cs42l73";
    model = "cs42l73";
    cpu-dai = <&ssi1>;
    audio-codec = <&codec>;
    audio-routing =
    "MIC_IN", "Mic Jack",
    "Mic Jack", "Mic Bias",
    "Headphone Jack", "HP_OUT";
    mux-int-port = <1>;
    mux-ext-port = <3>;
};
lcd@0 {
    compatible = "fsl,lcd";
    ipu_id = <0>;
    disp_id = <0>;
    default_ifmt = "RGB24";
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_ipu1>;
    status = "okay";
};
```

Şekil 8. Tasarlanan sistemin aygıt örnekleri

Gözlemlediğimiz kadarıyla DT yapısı aygıtların birbirleri ile olan hiyerarşik ilişkilerini tanımlamak için oldukça başarılıdır. Bununla birlikte sistemlerde bulunan kesme sinyalleri karmaşık yapılardır. Örneğin bir seri hattını sistemdeki bir veri yolunun çocuk düğümü olarak tanımlamak mantıklıdır. Bununla birlikte kesme özellikleri içeren bu seri hattını kesme denetleyicisinin çocuk düğümü olarak tanımlamak da uygun görünmektedir. Bunun gibi karmaşık durumlarda aygıtların tanımlanacağı düğümler olarak birincil adresleme ve kontrol özelliklerinin bulunduğu düğümler kullanılmaktadır. İkincil bağlantılar ise *phandle* adı verilen bir bağlantı ile tanımlanmaktadır. *phandle* bir düğümden diğer düğüme işaret eden bir niteliktir. Kesme yapısının uygulanabilmesi için kesme kontrolünde *interrupt-parent* ve *interrupts* nitelikleri kullanılmaktadır. *interrupt-parent*, kesme denetleyicisini açıklayan düğümün bir *phandle*'ıdır. Kesme denetleyicisindeki kesme sinyallerinin listesi *interrupts* niteliğinde bulunmaktadır. Kesme denetleyicisi düğümleri, *interrupt-controller* adı verilen boş bir niteliği tanımlamaktadır. Ayrıca bu düğümler *#interrupt-cells* ve *#address-cells* niteliklerini de tanımlamaktadır. *#interrupt-cells*, kesme denetleyicisinde tek bir kesme sinyali belirtmek için gereken hücre sayısı olarak açıklanabilir. Ayrıca, *#address-cells*, bir adres değeri için gereken hücre sayısını belirtmektedir. Şekil 9'da bu çalışmada SPI ara yüzü kullanılarak çalıştırılan bir Genel Amaçlı Giriş-Çıkış bacağı (ing. General Purpose Input-Output) (GPIO) arttırıcısının düğümü verilmiştir. Burada bazı bacakların kesme özelliğinin olması istenmektedir. Bu istekleri karşılamak için *#interrupt-parent* niteliğinde *gpio6* şeklinde arama yapılmıştır. Böylece bu GPIO arttırıcı aygıtının bacakları artık kesme olarak kullanılabilmektedir.

```
gpio@0 {
    compatible = "microchip,mcp23s08";
    #gpio-cells = <1>;
    gpio-controller;
    reg = <0>;
    mcp,spi-present-mask = <0x03>;
    mcp,gpio-base = <300>;
    spi-max-frequency = <10000000>;
    #interrupt-controller;
    #interrupt-parent = <&gpio6>;
    #interrupts = <11 4>;
    #interrupt-cells = <2>;
};
```

Şekil 9. Tasarlanan sistemde kesme yapısı örneği

Sistemin işletim sistemlerinde aygıt aramasını kolaylaştırmak için genellikle bir *aliases* düğümü tanımlanmaktadır. Bu nitelik, arama sırasında tam yolu belirtmek zorunda kalmadan bir aygıtı tanımlamak için kısa metot sağlamaktadır. Bu yöntem kart üzerinde bulunan, daha genel kullanım amaçlı olan aygıtlar için kullanılmaktadır. Aşağıda *aliases* düğümünün kullanım örnekleri verilmiştir.

```
/ {
    aliases {
        gpio0 = &gpio1;
        gpio1 = &gpio2;
        mmc0 = &usdhc1;
        serial0 = &uart1;
        spi0 = &ecspi1;
        spi1 = &ecspi2;
    };
```

Şekil 10. Tasarlanan sistemde aliases örneği

IV. TARTIŞMA

Çalışmada, bir mikro işlemci tabanlı gömülü sistem için DT kullanılarak sistemin ihtiyaç duyduğu yazılımların geliştirilmesi yerine getirilmiştir. Gömülü sistemin işlemci, bellek, USB, sıvı kristal ekran (ing. liquid crystal display) LCD), I2C, GPIO, darbe genişlik modülasyonu (ing. pulse width modulation) (PWM), SPI gibi birçok ara yüzü ve özelliğine ait yazılımlar doğrudan DT yapısı kullanılarak hazırlanmıştır ve geliştirilen gömülü yazılımların başarıyla çalıştıkları gözlemlenmiştir. Tüm özellikler Linux üzerinde geliştirilen küçük test yazılımları ile denenmiştir. Fonksiyonellik ve aygıt performansı açısından herhangi bir problem ile karşılaşılmamıştır. Klasik yöntem olan işletim sisteminin çekirdeğinde bulunan sürücü, .c uzantılı, kod sayfalarında yazılım geliştirme yöntemine göre bu bildiride tanıtılan DT yapısına dayalı yazılım geliştirme yöntemi karşılaştırılmıştır. DT kullanımında klasik/mevcut yazılım geliştirme yöntemlerinde sıklıkla karşılaşılan işletim sistemi çekirdeğinin sürekli değiştirilmesi ve derlenmesi probleminin ortadan kalktığı görülmüştür. Bu durum sistemin geliştirme

çalışmalarını oldukça hızlandırmakta ve basitleştirmektedir. Klasik yöntemde bir gömülü sistem yazılımı geliştirme çalışmaları 6-7 ay civarında sürmekte iken DT yapısı kullanarak bu sürenin 2-3 aya kadar düştüğü görülmüştür. Çalışmaların İzmir'de bulunan ve özellikle toplu taşıma araçları başta olmak üzere çeşitli sektörlerde elektronik ücret toplama, araç takibi, yolcu bilgilendirme ve araç içi kamera kayıt hizmetleri için yazılım ve donanım sistemleri üreten Kent Kart Ege Elektronik A.Ş. [14] Ar-Ge merkezinde yapılma fırsatı bulunmuştur. Elde edilen geliştirme süreleri, başta toplu taşıma ücretlendirme ve akıllı ulaşım alanları olmak üzere çeşitli uygulama alanları için gömülü sistem yazılımlarının geliştirilmesi çalışmalarımız sırasında gözlemlenmiştir. Ayrıca DT tabanlı yazılım geliştirmenin, geliştiricilerin faklı işletim sistemi çalışma mantıklarını ve özel programlama dillerini derinlemesine öğrenme zorunluluğunu da büyük ölçüde ortadan kaldırdığı bu çalışma ile tecrübe edilmiştir. Örneğin bu bildiride tanıtılan DT tabanlı yazılım geliştirme süreci Kent Kart A.Ş. bünyesinde çift çekirdekli iMX6 işlemcisi kullanılan bir sürücü bilgisayarı sisteminin geliştirilmesinde denenmiştir. Sürücü bilgisayarında, LCD, dokunmatik ekran, RF kart, yüksek çözünürlüklü çoklu ortam ara yüzü (ing. High Definition Multimedia Interface) (HDMI), güvenli sayısal hafıza kartı (ing. Secure Digital Memory Card) (SD), kullanıcı kimlikleme kartı (ing. Subscriber Identification Module) (SIM), küresel konumlama sistemi (ing. Global Positioning System) (GPS), mobil iletişim için küresel Sistem (ing. Global System for Mobile Communications) (GSM), USB, GPIO, kablosuz bağlantı alanı (ing. Wireless Fidelity) (Wi-Fi) gibi birçok kullanıcı özelliği DT kullanılarak başarılı bir şekilde çalıştırılmıştır.

Öte yandan gerçekleştirdiğimiz çalışmada her ne kadar DT yapısı, çekirdek kaynak koduna dokunmadan çevresel işlemlerin yapılmasını sağlasa da özellikle geniş yelpazedeki çeşitli mikro işlemci mimarilerinde çalışacak sistemlerin geliştirilmesinde DT yapısının uygulanmasının alışılagelmiş programlama tekniğinden farklı bir yapıya sahip olması, DT'nin geliştiriciler tarafından öğrenmesinin zaman alması ve farklı platformlar için konfigürasyonda değişiklik yapılması zorunlulukları nedeniyle bazı durumlarda dezavantajlı olabileceği gözlemlenmiştir.

V. Sonuç

Mikro işlemcili gömülü sistemler üzerinde çalışacak yazılımların DT yapısı kullanılarak tasarımı ve uygulanmasına yönelik bir çalışma gerçekleştirilmiştir. DT kullanılarak geliştirilen gömülü sistemin tüm özelliklerinin başarıyla çalıştırılabildiği gözlemlenmiştir. Klasik gömülü sistem geliştirme çalışması ile DT tabanlı yazılım geliştirme çalışması karşılaştırılmıştır. Yazılım geliştirmede DT yapısının kullanılmasının klasik yönteme göre daha basit ve hızlı olduğu görülmüştür.

Ancak, DT kullanımının geliştiriciler için yeni bir sözdizim öğrenme ve farklı bir yazılım geliştirme sürecine adapte olma gibi bazı zorlukları da beraberinde getirdiği tespit edilmiştir.

Bu zorlukların giderilmesi için model güdümlü geliştirmeye dayalı yeni yazılım geliştirme yöntemlerinin ve araç kümelerinin önemli bir çözüm olabileceğine inanılmaktadır. Gelecek çalışmamızda DT yazılımları için sözü edilen model güdümlü geliştirmeyi sağlayacak bir yöntemin ve bunu destekleyen yazılım mimarilerinin geliştirilmesi hedeflenmektedir.

Teşekkür

Yazarlar, bu çalışma için donanım ve yazılım kaynaklarının kullanılmasını sağlayarak destek veren Kent Kart Ege Elektronik A.Ş.'ye ve çalışmaların değerlendirilmesinde geribildirimleri ile destek olan Kent Kart Ar-Ge merkezi mühendislerine teşekkür eder.

Kaynakça

[1] D.D. Gajski, F. Vahid, S. Narayan and J. Gong, Specification and Design of Embedded Systems, 1st ed., Prentice Hall, NJ: Upper Saddle River, 1994.

[2] S. Arslan, M. Gündüzalp and E. Türk, "A multimedia application for an embedded system," National Conference on Electrical, Electronics and Biomedical Engineering, pp.189-193, 2016.

[3] T. Petazonni, "Device Tree," Free Electrons, Kernel, drivers and embedded Linux development, 2013.

[4] C. Simmonds, Mastering Embedded Linux Programming, UK: Birmingham, 2015.

[5] Devicetree Community, "The Devicetree Specification", 2016.

[6] G. Likely and J. Boyer, "A Symphony of Flavours: Using the device tree to describe embedded hardware," Linux Symposium, vol. 2, pp. 27-37, 2008.

[7] C. Devigne, J.B. Brejon, Q.L. Meunier and F. Wajsbürt, "Executing secured virtual machines within a manycore architecture," Microprocessors and Microsystems, vol. 48, pp. 21-35, 2017.

[8] B. Nikkel, "NVM express drives and digital forensics," Digital Investigation, vol. 16, pp. 38-45, 2016.

[9] G. Nicolescu and P.J. Mosterman, Model-based Design for Embedded Systems. United States of America: Taylor & Francis Group, 2010.

[10] M. Jassi, U. Sharif, D. Müller-Gritschneder and U. Schlichtmann, "Hardware-accelerated software library drivers generation for IP-centric SoC designs," The 26th Great Lakes Symposium on VLSI, pp. 287-292, 2016.

[11] IEEE Standard, 1685-2014 - IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows, 2014.

[12] NXP Semiconductor, i.MX 6Dual/6Quad Applications Processor Reference Manual, IMX6DQRM Rev. 1, 04/2013.

[13] IEEE Standart, 1275-1994 - IEEE Standard for Boot (Initialization Configuration) Firmware: Core Requirements and Practices, 1994.

[14] Kentkart Elektronik Ücret Toplama Sistemleri, www.kentkart.com/, 2017.