

Utilisation d'une sonde JTAG pour le développement embarqué

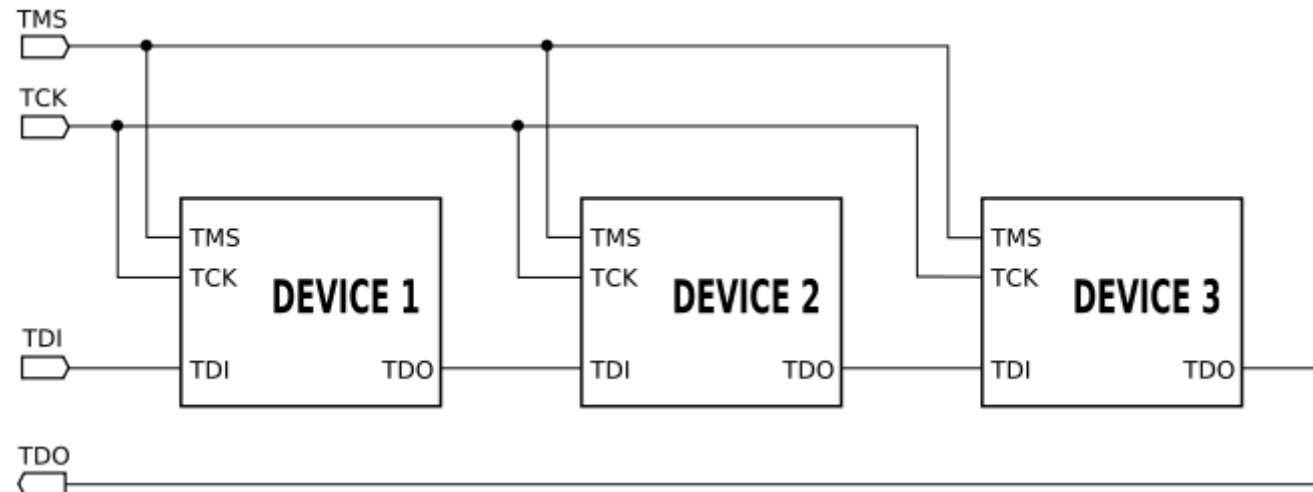
Julien Aubé
Julien.aube@openwide.fr

avril 2011

- Description
- Utilisations
 - Démarrage de carte, prototypage
 - Flashage de bootloader
 - Debug du chargement de uboot
 - Debug de linux
 - Recherche de bug kernel par bisection


- 1990 Join Test Action Group
 - D'abord pour le Boundary Scan
 - Puis utilisation pour le debug : Lecture/écriture en RAM/EEPROM/Flash, points d'arrêt matériels, modification des registres, de la MMU, du pointeur d'instruction
 - Plus récemment utilisation pour le profilage non-intrusif, statistiques sur l'exécution du code, et même la température interne !
- Plateformes :
 - FPGA, CPLD
 - Processeurs : ARM/MIPS/x86/PowerPC...

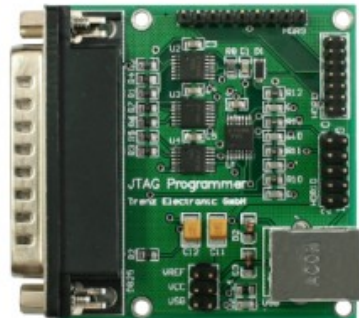
- Interface Test Access Port :
Bus série synchrone 4 fils multi-périphériques (parfois internes à un seul composant)
- Basé sur une machine à état munie de plusieurs registres de tailles variables.
- Quelques registres obligatoires (IDCODE, BYPASS)
- La plupart optionnels, non normalisés



- Utilisation et extension du port TAP pour le flashage, debug et profilage, via des extensions:
 - ARM : EmbeddedICE, CoreSight, *Nexus*
 - MIPS : EJTAG
 - PowerPC : RiscWatch (IBM),
 - FPGA, Atmel AVR/SAM,...
- Différences entre processeurs :
 - Sur la possibilité & manière d'accès aux registres, mémoire, flash...
 - Sur l'étendue des possibilités offertes
 - Sur la documentation du processeur à ce sujet

Sondes JTAG

- Une par fondeur de puce, logiciel propriétaire 
- Quelques sondes *universelles*, logiciel propriétaire
- Sondes USB « simples »



- **Attention au voltage !**

- Carte en général inspirée d'un « reference design »
- Les problèmes arrivent lorsque le reference design n'est pas 100% suivi : modification des puces RAM, mode d'horloge différent, flash différente...
- Le JTAG permet des tests rapides sur les toutes premières instructions machine, en bloquant le processeur avant le premier *Instruction Fetch*.
- Permet aussi de tester rapidement la plupart des fonctions, pour identifier des problèmes durant le prototypage

```
# Marvell SheevaPlug
proc plug_init { } {
  jtag_reset 0 1
  halt 0
  jtag_reset 0 0
  wait_halt
  [...]
  arm mcr 15 0 0 1 0 0x00052078
  mww 0xD0001400 0x43000C30 ;# DDR SDRAM Configuration
  mww 0xD0001404 0x39543000 ;# Dunit Control Low
  mww 0xD0001408 0x22125451 ;# DDR SDRAM Timing (Low)
  mww 0xD000140C 0x00000833 ;# DDR SDRAM Timing (High)
  mww 0xD0001410 0x000000CC ;# DDR SDRAM Address Control
  mww 0xD0001414 0x00000000 ;# DDR SDRAM Open Pages Control
  mww 0xD0001418 0x00000000 ;# DDR SDRAM Operation
  mww 0xD000141C 0x00000C52 ;# DDR SDRAM Mode
  mww 0xD0001420 0x00000042 ;# DDR SDRAM Extended Mode
  mww 0xD0001424 0x0000F17F ;# Dunit Control High
  mww 0xD0001428 0x00085520 ;# Dunit Control High
```

Flashage

- L'un des usages les plus importants du JTAG
- Plusieurs méthodes :
 - Accès direct au bus de la flash
 - Accès en chargeant les données en RAM, puis en chargeant un programme dans le CPU qui va écrire ces données en Flash
 - Utilisation du Boundary Scan
- Utilisation d'outils constructeurs/propriétaires : pas toujours possible, ex : changement du type de flash ou de bus
- Utilisation d'outils libres : **URJtag** et **OpenOCD**

- Outil simple, basé sur des fichiers de description BSDL
- Permet seulement de flasher, et de lire / écrire sur des adresses physiques
- Dispose d'un nombre important de « méthodes d'accès », c'est-à-dire de supports pour les bus spécifiques de chaque processeur
- Nécessite parfois une adaptation dans le code source, (ex : support d'un modèle de flash spécifique)
- Scriptable : Permet d'automatiser la procédure



```
jtag> detect
longueur IR: 5
Longueur de la chaîne: 1
Device Id: 00000110001101011000000101111111 (0x000000000635817F)
  Manufacturier: Broadcom
  Part(0):      BCM6358
  Pas      :    V1
  nom de fichier: /usr/local/share/urjtag/broadcom/bcm6358/bcm6358
jtag> endian big
jtag> initbus ejtag
ImpCode=00000000100000011000100100000100 00818904
EJTAG version: <= 2.0
EJTAG Implementation flags: R4k MIPS16 DMA MIPS32
Clear memory protection bit in DCR
Clear Watchdog
Potential flash base address: [0x0], [0x1e00000c]
Processor successfully switched in debug mode.
jtag> detectflash
jtag> eraseflash 0x1E000000 1
jtag> flashmem 0x1E000000 /home/jaube/Desktop/Downloads/cfe.dump
```



- Un outil plus complexe à maîtriser
- Basé sur un langage de script TCL
- Embarqué dans bon nombre de sondes professionnelles
- Permet :
 - Flashage
 - Debug direct (ligne de commande, script)
 - Backend GDB

- Tout est script dans OpenOCD
 - Description des cartes / cpu / flash / sondes dans /usr/share/openocd/scripts/
- Utilisation de ligne de commande :
 - telnet localhost 4444
- Pour flasher :
 - Définir le fichier de configuration de la plateforme (prendre un modèle existant, puis décrire la nouvelle carte en terme de RAM, flash,...)
 - Faire une séquence *halt-reset* pour placer le processeur en état stoppé juste après un reset
 - Utiliser *nand probe* puis *nand erase/nand write*, ou bien *flash probe* puis *flash erase_sector/flash write_image*.

Exemple sur Marvell Sheevaplug

```
> reset halt
target state: halted
target halted in ARM state due to debug-request, current mode:
Supervisor
cpsr: 0x400000d3 pc: 0x0060a608
MMU: disabled, D-Cache: disabled, I-Cache: enabled
> nand probe 0
NAND flash device 'NAND 512MiB 3.3V 8-bit (Samsung)' found
> nand erase 0 0 131072
erased blocks 0 to 1 on NAND flash device #0 'NAND 512MiB 3.3V 8-bit'
> nand write 0 plug_uboot.bin 0
.... wait ....
>
```

Debug du bootloader

- OpenOCD s'utilise comme un back-end GDB
- Permet de debugger le bootloader dès les premières instructions
- Permet de debugger par exemple pendant le développement du pilote réseau du bootloader
- Permet de debugger aussi le tout début de l'exécution du kernel Linux
- Debug sur le code de production

- Pour debugger uboot, on va charger une image ELF en RAM après initialisation grâce à une fonction openOCD:

- `plug_load_uboot()` :

```
plug_init
```

```
load_image uboot.elf
```

```
verify_image uboot.elf
```

```
bp 0x00600050 4 hw <----- Ajout d'un breakpoint
```

```
resume 0x00600000
```

```
237096 bytes written at address 0x00600000
```

```
downloaded 237096 bytes in 1.194556s (193.829 KiB/s)
```

```
verified 237096 bytes in 0.862387s (268.486 KiB/s)
```

```
target state: halted
```

```
target halted in ARM state due to debug-request, current  
mode: Supervisor
```

```
cpsr: 0x600000d3 pc: 0x00600050
```

```
MMU: disabled, D-Cache: disabled, I-Cache: disabled
```

- Ensuite on charge GDB avec l'image ELF du binaire :

```
armv5tel-redhat-linux-gnueabi-gdb u-boot
```

```
(gdb) add-symbol-file u-boot 0x00600000
```

```
(gdb) target remote :3333
```

```
Remote debugging using localhost:3333
```

```
reset () at start.S:161
```

```
161    mrs r0,cpsr
```

```
Current language:  auto; currently asm
```

```
(gdb) info frame
```

```
Stack level 0, frame at 0x1fe55238:
```

```
pc = 0x600050 in reset (start.S:161); saved pc 0x1ff96b54
```

```
called by frame at 0x0
```

Debug de Linux (boot)

- Même principe que le bootloader
- On peut utiliser la méthode précédente pour debugger la procédure de récupération du noyau sur la flash, et la décompression.
- Lorsque le kernel démarre, il est plus simple de tout piloter avec GDB
- Uniquement utile lors du démarrage de carte: lorsqu'un kernel minimal fonctionne, il est plus simple d'utiliser le kgdb intégré au kernel linux.

```
(gdb) file vmlinux
(gdb) target remote :3333
(gdb) break __init_begin
(gdb) mon reset <--- redémarre la carte
      Breakpoint 1, 0xc0008000 in stext ()
(gdb) load vmlinux
      Loading section .text.head, size 0x240 lma 0xc0008000
      Loading section .init, size 0xe4dc0 lma 0xc0008240
      [... cette partie peut être longue ! ...]
      Loading section __ksymtab_strings, size 0xc040 lma 0xc030ebcc
      Loading section __param, size 0x2e4 lma 0xc031ac0c
      Loading section .data, size 0x1e76c lma 0xc031c000
      Start address 0xc0008000, load size 3345456
      Transfer rate: 64 KB/sec, 15632 bytes/write.
```

```
(gdb) cont
```

- Récupération du log kernel dans GDB
 - Définition d'une macro :

```
define dmesg
    set $__log_buf = $arg0
    set $log_start = $arg1
    set $log_end = $arg2
    set $x = $log_start
    echo "
    while ($x < $log_end)
        set $c = (char)(($__log_buf)[$x++])
        printf "%c" , $c
    end
    echo "\n
end
document dmesg
dmesg __log_buf log_start log_end
Print the content of the kernel message buffer
end
```

- Appel avec `dmesg __log_buf log_start log_end`

Bissection via JTAG

- En s'appuyant sur :
 - Possibilité d'automatiser le chargement et le boot du kernel,
 - Instrumentation externe de certaines valeurs des variables du kernel
- ➔ Utilisation dans un banc de test
- ➔ Génération de stimuli artificiels en « out-of-band »
- ➔ Tests de non-régressions automatisés (via le reset de la carte), donc de **bissection**

- Bissection : permet de découvrir quel commit (parmi un grand nombre) introduit un bug
- Lorsque ce bug bloque le chargement du noyau, il est parfois utile d'écrire un script de bissection qui pilote la sonde JTAG via OpenOCD
- Utilisation de `git bisect run` pour appeler un script
- Ce script peut par exemple appeler OpenOCD avec un fichier de paramètre en TCL
- La valeur de retour du script est 0 pour une passe réussie, autre chose pour une passe ratée.

- L'utilisation du JTAG est critique au démarrage d'un projet.
- Les logiciels propriétaires sont performants, mais souvent dédiés à une plateforme ou un outil
- OpenOCD est complexe, mais pas tellement plus que certains outils propriétaires
- OpenOCD ouvre de nouvelles possibilités de tests de non régression et d'automatisation

questions ?