BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF RADIO ELECTRONICS

ÚSTAV RADIOELEKTRONIKY

PROGRAM UPDATE OF ZYNQ-BASED DEVICES

AKTUALIZACE PROGRAMU V ZAŘÍZENÍ S OBVODY ZYNQ

MASTER'S THESIS

DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. Branislav Michálek**

AUTOR PRÁCE

SUPERVISOR — **Ing. Michal Kubíček, Ph.D.**

VEDOUCÍ PRÁCE

CONSULTANT — **Ing. Jan Král**

KONZULTANT

BRNO 2019

ABSTRACT

Among many which are placed on modern embedded systems is also the need of storing multiple system boot image versions and the ability to select from them upon boot time, depending on a function which they provide. This thesis describes the development of a system update application for Xilinx Zynq-7000 devices. The application includes a simple embedded HTTP server for a remote le transfer. A client is allowed to upload the boot image le with the system update from either command line application or using the web page developed for this purpose.

KEYWORDS

Zynq-7000, System on Chip, Booting, HHypertext Transfer Protocol

ABSTRAKT

Mezi mnohé poºadavky kladené na moderní vestav¥né systémy pat°í pot°eba uchovávat více verzí jejich systémového obrazu (rmware, software nebo hardwareová kon gurace) a také moºnost volby, který z t¥chto obraz· systém na£te b¥hem procesu bootování, v závislosti na funkci, kterou daný obraz poskytuje. Tato diplomová práce popisuje vývoj aplikace pro za°ízení s obvody Zynq rmy Xilinx, jejíº funkcí je provést aktualizaci systému. Aplikace zahrnuje jednoduchý vestav¥ný HTTP server slouºící ke vzdálenému p°enosu soubor·. Klientovi umoº¬uje nahrát soubor s obrazem skrze aplikaci spustitelnou z p°íkazové °ádky, nebo prost°ednictvím webové stránky, která byla navrºena k tomuto ú£elu.

KLÍƒOVÁ SLOVA

Zynq-7000, systém na £ipu, bootování, hypertextový p°enosový protokol

MICHÁLEK, Branislav. Program update of Zynq-based devices. Brno, Rok, 49 p. Master's Thesis. Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Radio Electronics. Advised by Ing. Michal Kubí£ek, Ph.D.

This thesis was typeset using the thesis package version 3.03 http://latex.feec.vutbr.cz

DECLARATION

I declare that I have written the Master's Thesis titled Program update of Zynq-based devices independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the thesis and listed in the comprehensive bibliography at the end of the thesis.

As the author I furthermore declare that, with respect to the creation of this Master's Thesis, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll., Section 2, Head VI, Part 4.

Brno ....

author's signature

ACKNOWLEDGEMENT

First and foremost, I would like to thank God Almighty for giving me the strength, knowledge, ability and opportunity to undertake the whole study. I would like to express my gratitude to my consultant Jan Král for the useful comments, remarks and engagement through the learning process of this master thesis. Furthermore I would like to thank my supervisor Michal Kubicek for his helpful advice concerning the study issues. I would like to thank all my loved ones, who have supported me throughout entire process, both by keeping me harmonious and helping me putting pieces together.

Brno . . . . . .

. . . . . .

author's signature

Faculty of Electrical Engineering
and Communication
Brno University of Technology
Purkynova 118, CZ-61200 Brno
Czech Republic
http://www.six.feec.vutbr.cz

ACKNOWLEDGEMENT

Research described in this Master's Thesis has been implemented in the laboratories supported by the SIX project; reg. no. CZ.1.05/2.1.00/03.0072, operational program Výzkum a vývoj pro inovace.

Brno ....

author's signature

Contents

Introduction 11

1 Zynq Devices Booting 12
1.1 Boot Devices and Boot Modes 13
1.2 Boot Stages 13
1.2.1 Stage 0 14
1.2.2 Stage 1 14
1.2.3 Stage 2 15
1.2.4 Multiboot 16
1.3 Boot Image Creation 16
1.3.1 Boot Image Layout 17

2 Remote File Transfer 19
2.1 Remote File Transfer Protocols 19
2.1.1 File Transfer Protocol 20
2.1.2 Trivial File Transfer Protocol 20
2.1.3 Secure Copy Protocol 21
2.1.4 Secure File Transfer Protocol 21
2.1.5 Hypertext Transfer Protocol 21

3 System Update Implementation 24
3.1 Target Device Description 24
3.2 System-Level Design and Considerations 25
3.3 Boot Image Building 27
3.4 Multiboot and Fallback Functionality 29
3.5 System Update Application 30
3.5.1 HTTP Server 31
3.5.2 Client Command-Line Application 33
3.5.3 Client Web Application 34
3.5.4 Boot Image Validation Process 35

4 Conclusion 37

Bibliography 38

List of symbols, physical constants and abbreviations 41

List of appendices 43

A Code Listings 44
A.1 JavaScript Function to Transfer a Boot Image 44
A.2 Python Script to Build a Boot Image 45
A.3 C Function for Boot Image Valiadtion 47

B Command Prompt Listings 49
B.1 cURL Client Command and Server Response 49

List of Figures

1.1 Zynq-7000 PS architectural overview [3]. 12
1.2 Boot image layout [6]. 18
3.1 SDR emulator - top view. 24
3.2 Flash memory address map. 26
3.3 System-level diagram of the proposed system update application. . . 27
3.4 Boot image building script owchart. 28
3.5 Multiboot functionality owchart. 29
3.6 Thread processing HTTP requests owchart. 31
3.7 HTTP response generation owchart. 33
3.8 A principal communication model using Ajax. 34
3.9 Layout of the system update webpage. 35

Introduction

A system update is a mechanism which ensures that a embedded device running an older version of the system runs with a more recent release when the update mechanism is done. This includes updating everything that de nes the system (bootloader, operation system kernel, software applications, hardware con guration, etc.), restarting running processes and eventually a reboot [1].

An ideal mechanism never ends up in an inconsistent state, always keeps the device usable (in case of update failure the device fallbacks to previous state or a recovery mode), minimizes downtime while updating, ensures integrity and security. A properly developed system update mechanism can reduce [2]:

- Reduce development cycle. The possibility of adding new functionality means all features do not need to be implemented in the original release.
- Fault management. A system update xes defects with a patch.
- Extend the life cycle of embedded device with updated software and hardware.

This thesis focuses on the design and implementation of the system update for Xilinx Zynq Single-on-Chip (SoC) devices. The rst chapter brie y introduces the Zynq devices booting stages, a boot image layout and its creation process. The second chapter describes some of the most used communication protocols used for remote le transfer.

The third chapter presents the system update application design and implementation, the device multiboot and fallback functionality and the process of creation of customized boot image le.

The last chapter summarizes the achieved results of the thesis and proposes suggestions for further development.

1 Zynq Devices Booting

Xilinx Zynq-7000 SoC is a processor-centric platform that o ers software, hardware and Input/Output (I/O) programmability). Zynq-7000 integrates the ARM based Processing System (PS) with 28 nm Programmable Logic (PL) in a single device.

The key component of the PS is Application Processor Unit (APU). The APU contains one or two identical Cortex-A9 Central Processing units (CPU), each with its own 32 KB instruction and data cache, a Floating-Point Unit (FPU) for acceleration of oating-point arithmetic and so-called NEON unit for multimedia operations. Each CPU also contains standard Memory Management Unit (MMU) which is needed in Linux-based operating systems. The architectural overview of the Zynq-7000 PS is depicted on Fig. 1.1.

Fig. 1.1: Zynq-7000 PS architectural overview [3].

The PS also includes 256 KB of shared On-Chip Memory (OCM) for general purpose, controller for external Double Data Rate (DDR) Random Access Memory (RAM), Non-Volatile Memory (NVM) interfaces supporting NAND, 8-bit parallel NOR, and up to two quad serial peripheral interface (QSPI) NOR ash devices.

Main communication peripherals include two Gigabyte Ethernet MAC modules, Universal Serial Bus (USB) 2.0, Universal Asynchronous Receiver-Transmitter (UART), Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C), Controller Area Network (CAN), and General-Purpose Input/Output (GPIO) controllers. Other custom application-speci c peripherals can be implemented in the PL, including complex hardware functions as digital signal processing (DSP), high performance computing (HPC) etc.

Main PL components include Con gurable Logic Blocks (CLB), 36 KB of dual-port block RAM, DSP slices, I/O blocks, serial data transceivers and two 12-bit Analog to Digital Converters (ADC). Each CLB contains 16 ip- ops, eight 6-input Lookup Tables (LUT) for programmable logic or distributed memory implementation, and two 4-bit adders for arithmetic functions.

1.1 Boot Devices and Boot Modes

A boot device is a ash memory device, which the system boots from. This is referred as a master boot mode [4]. Depending on selected boot device, there are four possible master boot modes:

- Quad-SPI Boot,
- NAND Boot,
- NOR Boot,
- SD Card Boot.

In QSPI boot mode, the boot device is a non-volatile serial NOR ash memory device, connected to a host system via the SPI. This mode supports multiple input and output (multi I/O) SPI devices with single bit (Single I/O) as well as optional two bit (Dual I/O) and four bit (Quad I/O) data bus width.

1.2 Boot Stages

Zynq-7000 devices use a multi-stage boot process. Both a non-secure and a secure boot is supported. For a secure boot, the PL must be powered on to enable the use of the security block located within the PL, which provides 256-bit Advanced Encryption Standard (AES) and Secure Hash Algorithm (SHA) decryption/authentication.

The PS hardware boot stages include power supply ramping, clocking, resets, pin strap sampling and the Phase-Locked Loop (PLL) initialization. Within 45 clock cycles of the PS reference clock, the hardware samples the seven boot mode strap pins and stores their settings in read-only registers.

1.2.1 Stage 0

A read-only BootROM code, located in the on-chip Read Only Memory (ROM), executes on the primary CPU after a system reset is performed. There are two basic types of reset - the Powe-On Reset (POR) which resets the whole system including all of the registers. All states, except those stored in the eFuse and Battery-Backed RAM (BBRAM) are lost. A non-POR causes the BootROM to execute, but the BootROM retains knowledge about the security level of the previous boot in the REBOOT_STATUS register. The non-POR sources include PS_SRST pin and internal system resets, also referred to as soft reset (e.g. software controlled reset, watchdog timer, etc.).

The PL power-up and initialization sequences can occur in parallel with or after the PS start-up. If the BootROM needs the PL powered up, then early in the BootROM execution the BootROM

The main tasks of the BootROM are to con gure the system, copy the rst partition (usually a primary bootloader) from the boot device to the OCM, and then branch the code execution to the OCM. Before the branch, the BootROM disables access to its ROM code.

The BootROM code reads the boot mode register to determine if a master or slave boot mode is used, and if master, also the type of boot device used. Then the BootROM code searches for a valid boot image header (also referred as the BootROM header). The header search continues until a valid header is found or the entire range has been searched. Invalid header is detected by calculating it's checksum.

In the secure boot mode, the BootROM has the ability to authenticate and decrypt the encrypted bootloader partition. Authenticating the bootloader partition the BootROM also validates its data integrity. In the non-secure boot mode, corruption in bootloader partition is not recognized [3].

1.2.2 Stage 1

This is generally a rst-stage bootloader (FSBL), but it can be any user-controlled code. As the stage 1 code is loaded to the OCM, its size is limited to 192 KB [3]. FSBL reads the partition header table of the boot image (see section 1.3.1) to nd a bitstream, second-stage bootloader (SSBL) or bare-metal application partition. Then it is responsible for:

- initialization of selected system features and peripherals (e.g. DDR memory, I/Os, system clock, etc.) with the PS con guration data,
- programming the PL using a bitstream, if provided,
- loading a SSBL or a bare-metal application code into DDR memory,

ˆ hando the execution to the SSBL or bare-metal application.

The FSBL also supports loading partitions from eMMC ash devices, con gured as a secondary boot device [4]. This is useful when there is a small QSPI ash which does not meet the memory requirements to store all the partitions. In this case, the primary boot mode needs to be set to QSPI (through the boot mode pins), but only the FSBL is placed on the QSPI ash. All the other partitions are on the eMMC ash instead. The FSBL ignores the con gured primary boot mode and loads the other partitions from eMMC.

If loading of any partition fails for some reason, the FSBL does a fallback and enables the BootROM to load another bootable image that is known to be in a good state, if such an image is present in the boot device. This image is often referred to as the golden image [4]. The FSBL updates a Multiboot Register and does a non-POR soft system reset, so that the BootROM executes again and loads the image pointed to by the Multiboot Register .

The FSBL does a fallback if any of the partitions is corrupted. A corrupted partition is detected either by performing an RSA authentication or calculating a md5 checksum.

1.2.3 Stage 2

During the stage 2 an application software is generally loaded, but it could be also a SSBL [4]. Booting larger systems (e.g. Linux-based operating systems) may require more complex procedures, which are unable to be performed by FSBL due to its own limitations. The SSBL is a special software which is able to properly load such a system and transfer the execution to it.

The Universal Boot Loader (U-boot) is open-source SSBL that is frequently used in embedded devices for booting Linux kernel, device tree, root le system and Linux applications [5]. Unlike the FSBL, U-boot typically runs in DDR, not OCM. Its features include the ability to load, decrypt, authenticate, and execute images from Ethernet, ash memory, SD cards and USB. Users are able to interact with U-boot and control the boot process using the built-in command interpreter.

As the U-boot's commands are fairly low-level, it takes several steps to boot a kernel, but this also makes U-boot more exible than other bootloaders, since the same commands can be used for more general tasks. Copying data with U-boot requires explicit speci cation of the physical memory addresses in destination memory [5].

1.2.4 Multiboot

Multiboot is a feature that allows the device to select from multiple boot images stored in a boot device [4]. The boot image which is to be loaded can be user-selectable, or it can be selected automatically by the system upon the user-de ned decision criteria. To select an image, the FSBL writes its base address divided by 32 KB into the MultibootRegister and then generates a non-POR system reset. When the BootROM executes after the reset, it looks for the boot image header pointed to by the MultibootRegister .

1.3 Boot Image Creation

A boot image is a binary le usually stored in a boot device which allows the associated hardware to boot. This may include the operating system, utilities and diagnostics, as well as boot and data recovery information. Once built, the boot image can be distributed to a target device, patched within reasonable limits, and remain disposable in case of a need.

The boot image is created by building the required boot image header, processing input data les and appending tables which describe them. An input le can be software, hardware con guration (bitstream), or general data [4]. All these les are referred to as*images* A software image is provided either in binary (BIN) or Executable and Linking Format (ELF) le format. Every image can have one or more *partitions*, e.g. an ELF le can have multiple loadable sections, each of which forms a partition in the boot image. There could be performed a chosen type of encryption and authentication on each partition.

Building the boot image in general involves the following steps [6]:

1. Creating a Boot Image Format (BIF) le.
2. Running the Xilinx Bootgen utility to create a bootable binary le.

The BIF le speci es every component of the boot image according to boot order. Several attributes can be applied to each component, e.g. the *bootloader* attribute applied to an ELF le identi es the le as the FSBL. The following code snippet is an example of a simple BIF le:

```
the_ROM_image:
    {
    [init]init\_data.int
    [bootloader]fsbl.elf
    bitstream.bit
    application.elf
    }
```

Bootgen is a Xilinx tool which merges image les together and builds a boot image. Bootgen comes with both a Graphical User Interface (GUI) and a command line option. The command line option can be scripted [6].

1.3.1 Boot Image Layout

Every boot image consists of the following components (see Fig. 1.2) [6]:

- a boot image header,
- a register initialization table,
- an image header table,
- a partition header table,
- a FSBL image,
- optionally other image partitions.

The boot image header (also referred to as the BootROM header) resides in the beginning of a boot image. The boot image header is a structure that contains information related to booting the FSBL. There is only one such structure in the entire boot image. This header is parsed by the BootROM to get determined, where the FSBL is stored in boot device and where it needs to be loaded in OCM. In case of secure boot, some encryption and authentication related parameters are stored in there. Boot image header also provides a User De ned Field (UDF). This 76 bytes of free space has potential to be used for storing boot image version and identi cation, time stamp, or other user de ned data. The UDF is written using the Bootgen. The input user de ned data is provided through a text le in the form of a hexadecimal string.

Immediately after the xed-sized boot image header (BootROM header) there is typically the image header table. It is a structure comprising a own header, followed by a linked list of image headers. The header of the image header table contains information which is common across all images, e.g. the total number of partitions present in the boot image. Each image header contains information, such as the image name, number of partitions associated with this image and pointer to a rst associated partition header.

The partition header table is an array of structures containing information and attributes related to each partition, such as partition size, address in boot device, load address in RAM, encryption and authentication related information, etc.

Fig. 1.2: Boot image layout [6].

2 Remote File Transfer

Both the PS and PL of Zynq-7000 are programmed using a conventional device driver, providing support for a user initiated update. In typical system update ow, the central site for developing and distributing software for elded embedded systems creates a boot image, which needs to be transferred to the system, typically using wired or wireless Ethernet. Initial destination for the new boot image is usually DDR RAM [2].

After the image is transferred to the device, a special software (either a user code or U-boot) copies the image from RAM to another location in RAM, OCM, or NVM (e.g. QSPI ash memory).

2.1 Remote File Transfer Protocols

File transfer is the transmission of a binary le over a telecommunication network from one system to another. Typically, le transfer is mediated by a le transfer communication protocol, which is a convention that describes how to transfer les between two endpoints.

Most of the protocols are designed for the Internet and its Internet protocol suite, commonly known as TCP/IP protocol stack [8]. Internet is a packet-switched network which transmits data divided into units called packets. A packet comprises of a header and a payload. Network applications make use of the services provided by the lower layers, especially the transport layer protocols: Transmission Control Protocol (TCP) or User Datagram Protocol (UDP), which provide reliable or unreliable pipes to other processes.

File transfer protocols are higher-level protocols that operate in the application layer of the TCP/IP stack [8]. File data is encapsulated into transport layer protocol units, such as TCP or UDP segments. The segments transmission is handled by lower layer protocols. Apart from a stream of bits from a le, a le transfer protocol unit may also contain some relevant metadata, such as the le name, le size, le attributes, etc. Some of the most widely used le transfer protocols are:

- File Transfer Protocol (FTP),
- Trivial File Transfer Protocol (TFTP),
- Secure Copy Protocol (SCP),
- Secure File Transfer Protocol (SFTP),
- Hypertext Transfer Protocol (HTTP).

2.1.1 File Transfer Protocol

FTP is a standard network protocol used for the transfer of les between a client and a server on a network. FTP users may authenticate themselves with a clear-text sign-in protocol, normally in the form of a username and password, or connect anonymously if the server is con gured to allow it [9].

In both, the active and passive modes of establishing the data connection, the client creates a TCP connection from a random, usually unprivileged port to the FTP server command port 21. The server responds over the control connection with three-digit status code in ASCII with an optional, human-readable explanation of request, e.g. "200 OK".

While transferring data over the network, four data representations can be used, from which most important are [9]:

- ASCII mode - inappropriate for les that contain data other then plain text.
- Image mode (also called binary mode) - the recipient stores the bytestream as the sending machine sends each le byte by byte.

Data transfer itself can be done in three modes [9]:

- Stream mode - the data is sent as a continuous stream, all processing is left to TCP. No terminator character is needed.
- Block mode - FTP breaks the data into several blocks comprising header, byte count, and data eld.
- Compressed mode - the data is compressed using a simple algorithm.

FTP does not encrypt its tra c. Usernames, passwords, commands and data can be read by anyone able to perform packet capture on the network. For secure transmission FTP may be secured either with Secure Sockets Layer (SSL) or Transport Layer Security (TLS) protocols or replaced with SFTP.

2.1.2 Trivial File Transfer Protocol

TFTP is a simple derivate of FTP which allows a client to get a le from or put a le onto a remote host [10]. It is mainly used for transferring rmware images and con guration les to network appliances like routers, rewalls, IP phones, etc. [11]. TFTP has been used for this purpose because it is very simple to be implemented by code with a small memory footprint. This is especially useful for low resourced Single-Board Computers (SBC) and SoCs.

TFTP uses UDP as its transport protocol. A transfer request is always initiated targeting port 69, but the data transfer ports are chose independently by the sender and receiver during the transfer initialization. Due to its simplicity, TFTP lacks most of the more advanced features o ered by more complex le transfer protocols.

TFTP only reads and writes les from or to a remote server. It cannot list, delete, or rename les or directories and it has no provision for user authentication [10].

2.1.3 Secure Copy Protocol

SCP is based on Secure Shell (SSH) protocol [12]. It was developed for secure transfer of les between a local and a remote host or between two remote hosts. It uses the same mechanism for authentication as SSH, thereby it ensures the authenticity and con dentiality of the data in transit. A client can send les to a server and also request les from a server. Normally, a client initiates an SSH connection to the remote host, and requests an SCP process to be started on the remote server [13]. The remote SCP process can operate in one of two modes:

- source mode, which reads les and sends them back to the client,
- sink mode, which accept the les sent by the client.

2.1.4 Secure File Transfer Protocol

SFTP (also referred to as SSH File Transfer Protocol) is an extension of the SSH protocol [12]. Compared to SCP, which only allows le transfer, SFTP allows for a range of management operations with remote les, including resuming interrupted transfers, directory listings, and remote le removal. While SCP is better designed for one-time le transfers between two networked endpoints, the SFTP does the same and adds the data management [13].

2.1.5 Hypertext Transfer Protocol

Hypertext Transfer Protocol is an application layer protocol designed within the framework of the Internet protocol suite [14]. HTTP usually uses TCP as underlying and reliable transport layer protocol. HTTP was developed to facilitate World Wide Web, where hypertext documents include hyperlinks to other resources that the user can access.

HTTP works as a request-response protocol in a client-server computing model [14]. A web browser may represent the client side, and an application running on a system hosting a website may represent the server side. The client submits a HTTP requestmessage to the server by establishing a TCP connection to a particular port on the server, typically port 80 [14]. The server waits for the client's request and returns a responsecomprising of a status line and a message. The body of this message is typically a requested resource. HTTP resource is identi ed and located on the server by a Uniform Resource Locator (URLs) [14].

The client and server communicate by sending plain-text ASCII messages. The request message consists of the following [14]:

- a request line,
- a request header elds,
- an empty line,
- an optional message body.

The request line and other header elds must all be terminated with Carriage Return (CR) and Line Feed (LF) characters [14].

HTTP de nes methods to indicate a desired action to be performed on a identi- ed resource. What this resource represents, whether pre-existing data or data that is generated dynamically, depends on the implementation of the server. The most used methods include [14]:

- GET - the GET method requestsa representation of the speci ed resource.
- HEAD - the HEAD method asks for a response identical to that of a GET request, but without the response body.
- POST - the POST method requests that the serveaccept the entity enclosed in the request. The POST data might be for example a block of data that is the result of submitting a web form to a data-handling process, or a le to be uploaded to the server.
- OPTIONS - the OPTIONS method returns the HTTP methods that the server supports for the speci ed URL.

Format of the response message is similar to the format of the request message. It contains a status line, which comprises a status code and reason message. HTTP response status codes are primarily divided into ve groups [14]:

- Informational 1xx,
- Successful 2xx,
- Redirection 3xx,
- Client Error 4xx,
- Server Error 5xx.

HTTP xes the bugs in FTP that make it inconvenient to use for many small transfers which are typical for web pages. FTP has a stateful control connection which maintains a current working directory, and each transfer requires a secondary connection through which data is transferred. HTTP is stateless and multiplexes control and data over a single connection from client to server on well-known port numbers, which if necessary, can trivially pass through Network Address Translation (NAT) gateways and is simple for rewalls to manage. When FTP is transferring over the data connection, the control connection is idle. If it takes too long to transfer a le, the rewall or NAT may decide that the control connection is dead and stop tracking it. This e ectively breaks the connection and confuses the transfer.

On the other hand, a single HTTP connection is only idle between requests and it is normal and expected for such connections to be dropped after a time-out.