Automatically finding undocumented ISP commands in the NXP LPC microcontroller family

Kevin Valk[1]
Supervisor: Roel Verdult[2]

[1] Radboud University
kevin@kevinvalk.nl
[2] Radboud University
rverdult@cs.ru.nl

Abstract

This research shows the importance of security checks on embedded systems, not only in the firmware but also the bootloader. Preliminary research on the NXP LPC 2148 microcontroller bootloader, shows that it contains an undocumented feature that can be used to bypass security. However, the undocumented feature has some basic protection, so it is not an immediate threat. This does raise the question, if there are more undocumented features inside the NXP LPC microcontroller family. This research shows to what extend this analyses can be automated. A Python script is developed for IDA to fully automate this analyses with a very generic approach, that should support as many different NXP LPC bootloaders, perhaps even all of them. Finally, the results section shows the output of the script on five different bootloaders from the NXP LPC family. All of these bootloaders contain the same undocumented feature. This shows that (semi-)automatic analyses of bootloaders are just as important as (semi-)automatic analyses of firmware in general.

1 Introduction

Embedded systems are computer systems that are designed for a specific goal inside a machine or electrical system. Embedded systems always have a hardware software co-design. A modern washing machine is a great example of an embedded system. Looking at the history of washing machines, it did not always had electronics components. However, nowadays it is hard to think of a washing machine that does not use electronics to turn on the water, rotate the drum, make beeps when its done and there are many more advanced features. These embedded devices are getting more and more integrated within our lives and the embedded devices also get smarter and more powerful every day. More importantly, with the upcoming of the internet of things, most of these embedded devices are connected to the internet [4]. It is not even a far fetched idea, to see washing machines hooked up to the internet, making it possible to turn on and control your washing machine from an app on your mobile phone.

In 2014, a study showed that many embedded systems, that are connected to the internet, had in there software one ore more vulnerabilities [6]. The news and the scientific community also frequently publishes stories about embedded systems with broken security or other privacy and security problems. This shows the severe state of security in embedded devices.

Every company has to make money of there products and is thus, inherit reluctant to share its secrets with others, even for security reviews. Thus, the need to test security in devices which are not open source, is very important. Especially, the parts of the software that will communicate with the outside world, bugs or exploits in these parts of the system can have severe consequences [13, 8]. Most of the time, incoming communication is interpreted through parsers, these parsers do not only have to be fool proof, but should never contain more then necessarily and it should obviously never contain backdoors.

Currently, there is research being done that focuses on (semi-)automatic Parser Identification and Analyses (PIA) in embedded systems [5]. However, embedded systems often contain a small piece of software which also communicates with the outside world, this is called a bootloader. A bootloader is responsible for the initial setup of the hardware and software and can be used to communicate with the chip on the lowest level. This can be used to update or install the firmware on the chip, debug the chip and other low level work on the chip. This shows that bootloaders do not differ from normal embedded software and should also be very secure as they not only communicate with the outside world, but can have full control over the embedded system. To investigate this hypotheses, preliminary research is required into how bootloaders work and how they communicated. This will show, to what extend these bootloaders should also be included in research into (semi-)automatic security analyses in embedded systems [9, 11, 5, 7, 12].

To this end, this research will focus on analysing a bootloader. The results of this research can then be used in the research into PIA [5]. As a starting point, the microcontroller family called LPC from NXP is chosen [3]. The LPC family is chosen because all chips use bootloaders that can communicate with the outside world and the LPC family currently exists out of more then 250 microcontrollers.

1.1 LPC2148

We had access to the BlueBoard-LPC214x which can been seen in figure 1. The manufacturer of this board is NGX Technologies[1] and the microcontroller controlling this board is the LPC2148 from NXP [10].

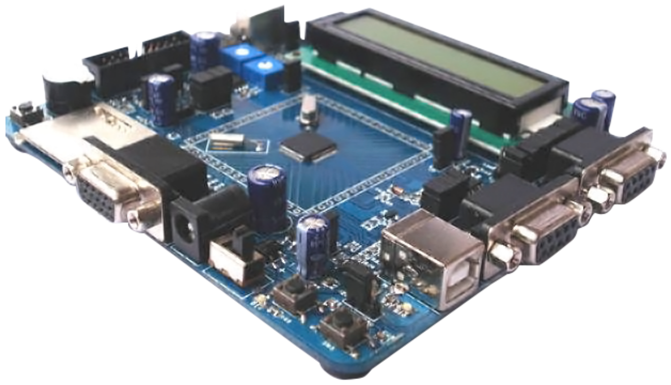


Figure 1: BlueBoard-LPC214x from NXP

The ease of access to this board made it a perfect target for the research on the bootloader. The bootloader is 12 kB in size and can be found remapped from the on-chip flash memory at address `0x7FFFD000` to `0x80000000`. It was a trivial task to dump the bootloader through JTAG.

While the LPC2148 is extensively documented with a 354 page document [10], the bootloader itself is not an open source project. Because of this, we reversed the bootloader according the documentation and looked for any discrepancies.

1.1.1 Reversing

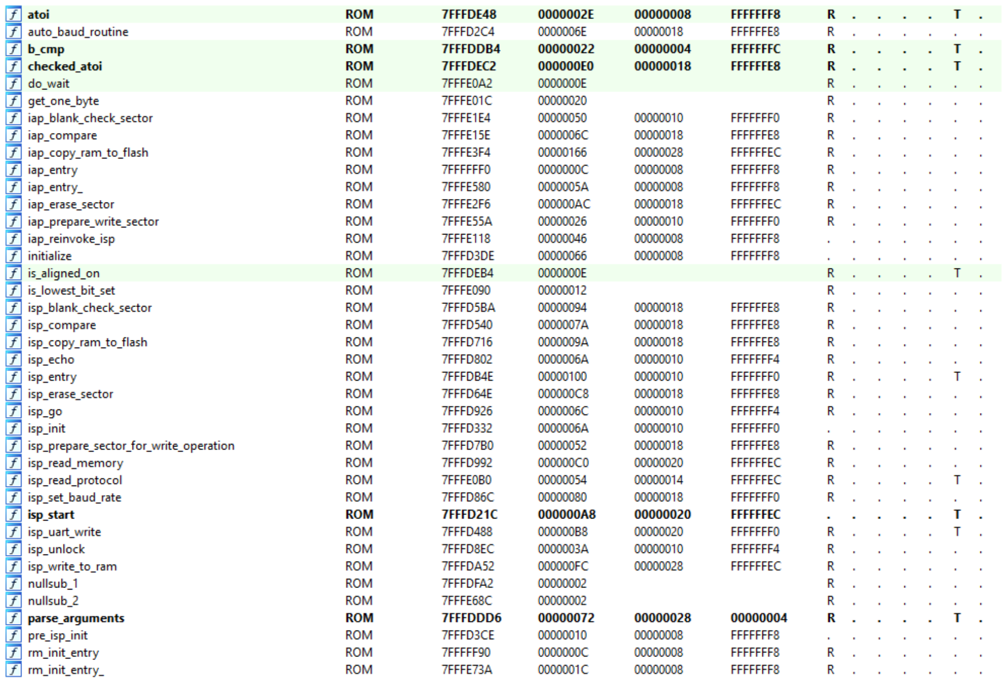
The LPC2148 bootloader (in this section, referred to as "bootloader") is reversed with IDA Pro [2] in a static fashion, so no dynamic analyses is done. The bootloader exists out of 106 functions, most of functions are rather small. To better facilitate reversing, all known memory segments were mapped in IDA from the documentation, this can be found in chapter 2 from the documentation.

According to the boot process flowchart (section 21.4.14 in the documentation), the *Code Read Protection (CRP)* is checked early in the boot process. The address of the CRP is publicly known, namely address `0x000001FC` and there are three external references to this CRP address. The second external reference is inside the entry point function (referred to as "initialize" function). This was checked by comparing the logic of the initialize function to the first steps of the boot process flowchart and the fact, that the initialize function itself, was never called within the bootloader. This made it clear that the initialize function was indeed the entrypoint of the LPC2148 microcontroller.

From this point on, it was easy to map the boot process flowchart inside the bootloader binary by using external references, access to memory segments and RAM access. This process

[1] http://shop.ngxtechnologies.com/product_info.php?products_id=28

was continued by reversing more functions and documenting them. A partial list of reversed functions can been seen in figure 2.



atoi	**ROM**	**7FFFDE48**	**0000002E**	**00000008**	**FFFFFFF8**	**R . . . . . T .**
auto_baud_routine	ROM	7FFFD2C4	0000006E	00000018	FFFFFFE8	R . . . .
b_cmp	**ROM**	**7FFFDDB4**	**00000022**	**00000004**	**FFFFFFFC**	**R . . . . . T .**
checked_atoi	**ROM**	**7FFFDEC2**	**000000E0**	**00000018**	**FFFFFFE8**	**R . . . . . T .**
do_wait	ROM	7FFFE0A2	0000000E			R . . . .
get_one_byte	ROM	7FFFE01C	00000020			R . . . .
iap_blank_check_sector	ROM	7FFFE1E4	00000050	00000010	FFFFFFF0	R . . . .
iap_compare	ROM	7FFFE15E	0000006C	00000018	FFFFFFE8	R . . . .
iap_copy_ram_to_flash	ROM	7FFFE3F4	00000166	00000028	FFFFFFEC	R . . . .
iap_entry	ROM	7FFFFFF0	0000000C	00000008	FFFFFFF8	R . . . .
iap_entry_	ROM	7FFFE580	0000005A	00000008	FFFFFFF8	R . . . .
iap_erase_sector	ROM	7FFFE2F6	000000AC	00000018	FFFFFFEC	R . . . .
iap_prepare_write_sector	ROM	7FFFE55A	00000026	00000010	FFFFFFF0	R . . . .
iap_reinvoke_isp	ROM	7FFFE118	00000046	00000008	FFFFFFF8	. . . .
initialize	ROM	7FFFD3DE	00000066	00000008	FFFFFFF8	. . . .
is_aligned_on	ROM	7FFFDEB4	0000000E			R . . . . . T .
is_lowest_bit_set	ROM	7FFFE090	00000012			R . . . .
isp_blank_check_sector	ROM	7FFFD5BA	00000094	00000018	FFFFFFE8	R . . . .
isp_compare	ROM	7FFFD540	0000007A	00000018	FFFFFFE8	R . . . .
isp_copy_ram_to_flash	ROM	7FFFD716	0000009A	00000018	FFFFFFE8	R . . . .
isp_echo	ROM	7FFFD802	0000006A	00000010	FFFFFFF4	R . . . .
isp_entry	ROM	7FFFDB4E	00000100	00000010	FFFFFFF0	R . . . . . T .
isp_erase_sector	ROM	7FFFD64E	000000C8	00000018	FFFFFFE8	R . . . .
isp_go	ROM	7FFFD926	0000006C	00000010	FFFFFFF4	R . . . .
isp_init	ROM	7FFFD332	0000006A	00000010	FFFFFFF0	. . . .
isp_prepare_sector_for_write_operation	ROM	7FFFD7B0	00000052	00000018	FFFFFFE8	R . . . .
isp_read_memory	ROM	7FFFD992	000000C0	00000020	FFFFFFEC	R . . . .
isp_read_protocol	ROM	7FFFE0B0	00000054	00000014	FFFFFFEC	R . . . . . T .
isp_set_baud_rate	ROM	7FFFD86C	00000080	00000018	FFFFFFF0	R . . . .
isp_start	**ROM**	**7FFFD21C**	**000000A8**	**00000020**	**FFFFFFEC**	**. . . . . . T .**
isp_uart_write	ROM	7FFFD488	000000B8	00000020	FFFFFFF0	R . . . . . T .
isp_unlock	ROM	7FFFD8EC	0000003A	00000010	FFFFFFF4	R . . . .
isp_write_to_ram	ROM	7FFFDA52	000000FC	00000028	FFFFFFEC	R . . . .
nullsub_1	ROM	7FFFDFA2	00000002			R . . . .
nullsub_2	ROM	7FFFE68C	00000002			R . . . .
parse_arguments	**ROM**	**7FFFDDD6**	**00000072**	**00000028**	**00000004**	**R . . . . . T .**
pre_isp_init	ROM	7FFFD3CE	00000010	00000008	FFFFFFF8	. . . .
rm_init_entry	ROM	7FFFFF90	0000000C	00000008	FFFFFFF8	R . . . .
rm_init_entry_	ROM	7FFFE73A	0000001C	00000008	FFFFFFF8	R . . . .

Figure 2: A partial list of reversed function from the LPC2148 bootloader

Inside the bootloader code, there was no real strange behaviour, aside from some strange compiler optimization and four variables that where stored in reversed memory. However, inside the *In System Programmer (ISP)* command handler there was an undocumented command.

1.2 In System Programmer

According to the LPC2148 documentation, the definition of an ISP is, "ISP is programming or reprogramming the on-chip flash memory, using the boot loader software and a serial port. This can be done when the part resides in the end-user board" [10] and this definition holds for any general ISP. It is not possible to change the embedded bootloader in the LPC microcontrollers, meaning, that the bootloader is also available on all systems in production. This makes the ISP in general a very interesting target. Certainly because of the preliminary research shows that there is an undocumented command inside the bootloader. The ISP is available on virtually all LPC microcontroller bootloaders and we hypothesize, that, bootloaders in the LPC family will be very similar, but further research has to point this out.

ISP Command	Usage
Unlock	`U <Unlock Code>`
Set Baud Rate	`B <Baud Rate> <stop bit>`
Echo	`A <setting>`
Write to RAM	`W <start address> <number of bytes>`
Read Memory	`R <address> <number of bytes>`
Prepare sector for write	`P <start sector number> <end sector number>`
Copy RAM to Flash	`C <Flash address> <RAM address> <number of bytes>`
Go	`G <address> <Mode>`
Erase sector(s)	`E <start sector number> <end sector number>`
Blank check sector(s)	`I <start sector number> <end sector number>`
Read Part ID	`J`
Read Boot code version	`K`
Compare	`M <address1> <address2> <number of bytes>`

Table 1: LPC2148 supported ISP commandos according the documentation [10]

The LPC2148 should support exactly 13 ISP commands, the commands can be seen in table 1. However, in the reversed bootloader, there was another command, which we call "isp_gpio_write". This function can be called like `T <address> <number of bytes> <width> <strobe>` through the serial port. The function that is responsible for handling this command can be found in appendix A. In essence, the T command is exactly the same as the `W` command (write to RAM). However, `T` can also write to RAM at addresses from `0x40000000` to `0x40000200`, this can not be done with the normal write to RAM command. This first RAM segment, contains important variables that are being used by watchdog, protection, the ISP itself and other initialization sub routines to correctly boot the microcontroller. This could be abused to bypass protection or change the chip while this was not intended.

The impact of this undocumented feature is small, because of an extra layer of protection that is built in the ISP, namely, CRP. This prevents, some or all commandos to be executed. Do note, that this `T` command is only protected by a single if statement. This if statement is only ran once at boot. If one would glitch this single check, full access to the chip could be obtained by using the `T` command to remove the CRP all together.

1.3 Code Read Protection

The ISP itself is protected by a very minimalistic *Access Control List (ACL)* over all the ISP commands. CRP is a 4 byte value, that resides in the flash at address `0x000001FC`. The first thing the microcontroller does when it powers on, is to copy this value from flash into RAM at address `40000128`. There are four different values possible for the CRP, the value defines to what extend the ISP is available or if it is available at all. When CRP has another value then the possible CRP values, then there is effectively no CRP. The different values and there corresponding behaviour, can been seen in table 2.

Name	Pattern	State
NO_CRP	Arbitrary	Everything can be accessed
NO_ISP	`0x4E697370`	ISP can not be accessed
CRP1	`0x12345678`	JTAG is disabled and the ISP is in the following state: `W` Write to RAM can not access RAM below `0x40000200` `C` Copy RAM to flash can not write to sector 0 `E` Erase can erase Sector 0 only when all sectors are selected for erase `R` Read command is disabled `T` Write to RAM from GPIO command is disabled `G` Go command is disabled `M` Compare command is disabled
CRP2	`0x87654321`	JTAG is disabled and the ISP is in the following state: `E` Erase only allows erasure of all user sectors `R` Read command is disabled `T` Write to RAM from GPIO command is disabled `G` Go command is disabled `M` Compare command is disabled `W` Write to RAM is disabled `C` Copy RAM to flash is disabled
CRP3	`0x43218765`	JTAG is disabled and the ISP can not be entered if a valid user code is present in flash sector 0. If there is no valid user code present CRP3 behaves like CRP2.

Table 2: Code Read Protection levels according the documentation [10], augmented from the reversed bootloader

1.4 NPX LPC microcontroller family

According to the Flash Magic supported devices page, the NXP LPC family has more then 250 difference LPC microcontrollers [1]. Random sampling of the data sheets from the different microcontrollers, shows that all these chips contain some form of an ISP. This immediately raises the question, if all these ISP implementations contain the undocumented `T` command or other undocumented "features". This leads us to the main question we want to address in this paper:

To what extend is it possible to automatically find the different ISP commands inside a bootloader from the NXP LPC microcontroller family?

If there would be a way to automatically get all supported ISP commands from any NXP LPC family microcontroller, it would be trivial to compare the supported commands to the official documentation. This would make it easier to automatically analyse NXP LPC microcontrollers for possible backdoors or undocumented features. It also makes it possible to compare different LPC chips to each other. Further research into specific ISP command for security analyses is also much easier, as it one can find the correct location of the handler with the script.

2 Automated ISP analyses

ISP is supported way back to chips made in 2001 (LPC700). However, the first documentation that has extensive information about the ISP is found in the LPC900 series and up. From that moment on, all LPC uses the same encoding for there ISP communication, namely UU encode. This fact can be used for automatic searching, namely, all ISP UART communication ends with `<crlf>`.

2.1 Switch structure

The target we initial analysed is the LPC2148. The corresponding bootloader uses a switch statement to parse all difference possible ISP commands. Scanning for switch statements that only have capital letters as switch cases would be a trivial solution to our research question. However, there are many ways to implement switch statements and more importantly, compilers can optimise switch statements in many different ways. It can happen that the compiler rewrites switch statements into if statements, or sometimes even more advanced optimisation are done, for example using an array of function pointers to directly jump to the corresponding handler. This makes scanning for the switch structures not a viable strategy when trying to apply the scanner on other bootloaders.

2.2 Call flow analyses

Because of compiler optimisations and the thousands of different ways to write code, it would be better to create a strategy that does not scan on specific statements but exploits other information that hardly change. Looking at what functions call other functions and having a few reference points in the call, one can deduce a lot of information about the program. Information obtained from reversing a few different bootloaders, shows that this is a viable strategy to scan for the ISP handlers. The current strategy exists out of the following steps:

- Find all functions that first do something with `\r` and then with `\n`. Exactly what they do is irrelevant, so all operands are scanned for those specific values.
  - For the function that has the most external references, it is assumed it is the "uart0_puts" function.
- Find all functions that are referenced only once (this is another observation from the available bootloaders).
  - For all these functions, check if they call the "uart0_puts" function, if so, add them to a candidate list.
  - For all candidates, make a list of the most called functions, from the three first functions called in the candidate function, it is assumed that this is some sort of sanitize or argument getter function for the ISP handler.
  - For all candidates, check if they call this sanitize function, if so, add them to an ISP handler list.
  - The ISP handler list now contains all ISP handlers.

Doing this by hand is not a viable solution, so it is required to automate this task. To achieve this, IDAPython is used.

2.3 IDAPython

Automating the call flow analyses is highly desirable as it is not suited for humans. IDA Pro was used for our general reversing and IDA Pro also support automated analyses with Python through the so called IDAPython plugin[2]. This plugin makes it possible to use any and all functions that are used inside IDA from a Python script. Writing an Python script that does all the steps given in section 2.2 is trivial. In appendix B a possible analyser script can be seen, that uses this strategy to search for all ISP handlers.

3 Results

There was access to seven different bootloaders from the LPC family. The results of executing the Python script described in section 2.3 on the different bootloader can be seen in table 3.

LPC	Status	Note
LPC11U35	`FAIL`	Unable to correctly load the dump, was unable to find the ISP handler manually
LPC1114	`PASS`	All non trivial ISP handler functions found
LPC1347	`FAIL`	Exact same problem as the LPC11U35
LPC1768	`PASS`	All non trivial ISP handler functions found
LPC1778	`PASS`	All non trivial ISP handler functions found
LPC2138	`PASS`	All non trivial ISP handler functions found
LPC2148	`PASS`	All non trivial ISP handler functions found

Table 3: Results of scanning the different LPC bootloaders

The script performs really well in most of the bootloaders. However, it fails at the LPC11U35 and the LPC1347. Currently it is not clear why exactly but it looks like the problem is not in the script but in the bootloader dump itself. Manual reversing of the bootloader does not show any ISP handler logic, while the corresponding documentations do show that these microcontrollers support ISP in the same way the other chips do.

Name	Address
isp_gpio_write	`0x1FFF0634`
sub_1FFF06CC	`0x1FFF06CC`
sub_1FFF0740	`0x1FFF0740`
sub_1FFF07D0	`0x1FFF07D0`
sub_1FFF08B4	`0x1FFF08B4`
sub_1FFF0958	`0x1FFF0958`
sub_1FFF09A4	`0x1FFF09A4`
sub_1FFF09D6	`0x1FFF09D6`
sub_1FFF0A86	`0x1FFF0A86`
sub_1FFF0ABC	`0x1FFF0ABC`
sub_1FFF0B22	`0x1FFF0B22`
sub_1FFF0BEA	`0x1FFF0BEA`

Table 4: Output on the LPC1768

Name	Address
isp_gpio_write	`0x1FFF02A4`
sub_1FFF034E	`0x1FFF034E`
sub_1FFF03C0	`0x1FFF03C0`
sub_1FFF0450	`0x1FFF0450`
sub_1FFF0552	`0x1FFF0552`
sub_1FFF05F4	`0x1FFF05F4`
sub_1FFF063E	`0x1FFF063E`
sub_1FFF066E	`0x1FFF066E`
sub_1FFF071A	`0x1FFF071A`
sub_1FFF0752	`0x1FFF0752`
sub_1FFF07B4	`0x1FFF07B4`
sub_1FFF0868	`0x1FFF0868`

Table 5: Output on the LPC1768

[2] https://code.google.com/p/idapython/

Name	Address
isp_gpio_write	0x1FFF02A0
sub_1FFF034A	0x1FFF034A
sub_1FFF03BC	0x1FFF03BC
sub_1FFF044C	0x1FFF044C
sub_1FFF054E	0x1FFF054E
sub_1FFF05F2	0x1FFF05F2
sub_1FFF063C	0x1FFF063C
sub_1FFF066C	0x1FFF066C
sub_1FFF0716	0x1FFF0716
sub_1FFF074E	0x1FFF074E
sub_1FFF07B0	0x1FFF07B0
sub_1FFF087C	0x1FFF087C

Table 6: Output on the LPC1778

Name	Address
isp_gpio_write	0x1FFF0494
sub_1FFF054C	0x1FFF054C
sub_1FFF05C6	0x1FFF05C6
sub_1FFF065A	0x1FFF065A
sub_1FFF0722	0x1FFF0722
sub_1FFF07BC	0x1FFF07BC
sub_1FFF080E	0x1FFF080E
sub_1FFF0878	0x1FFF0878
sub_1FFF08F8	0x1FFF08F8
sub_1FFF0932	0x1FFF0932
sub_1FFF099E	0x1FFF099E
sub_1FFF0A5E	0x1FFF0A5E

Table 7: Output on the LPC2138

Name	Address
isp_gpio_write	0x7FFFD488
isp_compare	0x7FFFD540
isp_blank_check_sector	0x7FFFD5BA
isp_erase_sector	0x7FFFD64E
isp_copy_ram_to_flash	0x7FFFD716
isp_prepare_sector_for_write_operation	0x7FFFD7B0
isp_echo	0x7FFFD802
isp_set_baud_rate	0x7FFFD86C
isp_unlock	0x7FFFD8EC
isp_go	0x7FFFD926
isp_read_memory	0x7FFFD992
isp_write_to_ram	0x7FFFDA52

Table 8: Output on the LPC2148 after it was reversed

Tables 4, 5, 6 and 7 show the output of the script when running the script on the corresponding bootloaders. No preprocessing was done what so ever to get these results. Comparing these tables to table 8 shows no difference in the ISP handler apart from the normal changes in addresses, which are expected. Table 8 is the output of the script on the LPC2148 after reversing the that bootloader ISP manually. Checking if the undocumented `T` command was present was done, by manually inspecting the functions in the different bootloaders. The `isp_gpio_write` is present in each bootloader, while the function is not mentioned in the corresponding documents. This raises the idea that this undocumented command is present in more LPC bootloaders, further research has to point out exactly how many bootloaders are not build to specifications.

4 Conclusion

This research has shown that even the smallest and the lowest level of codes should be checked on security issues. The undocumented `T` command that has been found within the seven bootloaders from the NXP LPC microcontroller show this. In this case there is no big damage to current released products that uses these chips, as the `T` command has the same protection as similar commands. Still, there should never be commands in there that are not in the documentation as they serve no purpose and could sometimes be abused.

(Semi-)Automatic analyses should not only focus on firmware of embedded systems but also on there corresponding bootloaders. Because it turns out that the bootloader can also communicate with the outside world and that this preliminary research shows that these bootloaders can also contain bugs, undocumented features and perhaps even backdoors.

References

[1] Flash magic device overview. http://www.flashmagictool.com/supporteddevices.html.

[2] IDA Pro website. https://www.hex-rays.com/products/ida/.

[3] LPC microcontrollers website. http://www.nxp.com/products/microcontrollers/.

[4] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. Computer Network, 54(15):2787–2805, Oct 2010.

[5] L. Cojocar and R. Verdult. PIA: Parsers Identification and Analysis in Embedded Systems. to appear.

[6] A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, and S. Antipolis. A Large-Scale Analysis of the Security of Embedded Firmwares. In USENIX Security Symposium, August 2014. https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-costin.pdf.

[7] D. Davidson, B. Moench, S. Jha, and T. Ristenpart. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. pages 463–478, 2013.

[8] L. Duflot, Y.-A. Perez, and B. Morin. What if you can't trust your network card? In Proceedings of the 14th international conference on Recent Advances in Intrusion Detection, RAID'11, pages 378–397. Springer-Verlag, 2011.

[9] Y. Li, J. M. McCune, and A. Perrig. VIPER: verifying the integrity of PERipherals' firmware. In Proceedings of the 18th ACM conference on Computer and communications security, CCS '11, pages 3–16. ACM, 2011.

[10] NXP. LPC2148 ARM7 Based Microcontroller, 2005. http://www.keil.com/dd/chip/3880.htm.

[11] C. Wysopal and C. Eng. Static detection of application backdoors. Black Hat, Aug 2007.

[12] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In Network and Distributed System Security (NDSS) Symposium, NDSS 14, February 2014. http://s3.eurecom.fr/tools/avatar/.

[13] J. Zaddach, A. Kurmus, D. Balzarotti, E.-O. Blass, A. Francillon, T. Goodspeed, M. Gupta, and I. Koltsidas. Implementation and implications of a stealth hard-drive backdoor. In Proceedings of the 29th Annual Computer Security Applications Conference, pages 279–288. ACM, 2013. http://doi.acm.org/10.1145/2523649.2523661.

A Decompiled T command

```c
/**
 * ISP GPIO write commando for the LPC2148
 * T <address> <number of bytes> <width> <strobe>
 * - address: address to start writing to (0x40000000 < address < 0x40008000)
 * - number of bytes: the number of bytes to write to SRAM
 * - width: amount of bits to read from GPIO, 8 for bytes, 16 for words
 * - strobe: boolean should a strobe be used (PIN7 low and then high again)
 *
 * Reads from the GPIO IO pins a byte or a word and writes this to given
 *  address. The address is increased and this is done for a number of times.
 */
void __fastcall isp_gpio_write(__int64 init1, int init2, int init3)
{
  __int32 ret;              // r4@1
  _BYTE *ascii;             // r0@8
  unsigned int v5;          // r0@14
  unsigned int v6;          // r0@14
  unsigned int v7;          // r0@15
  unsigned int arg2_4[2];   // [sp+0h] [bp-20h]@1
  unsigned int width;       // [sp+8h] [bp-18h]@1
  unsigned int arg1;        // [sp+Ch] [bp-14h]@1

  // Arguments are unused
  *(_QWORD *)arg2_4 = init1;
  width = init2;
  arg1 = init3;
  // Arg2: How many bytes are read from GPIO
  ret = checked_atoi(arg_two, arg2_4, 0x64, init1) & 0xFF;
  if ( !ret ) {
    // Arg1: Address to write to
    ret = checked_atoi(arg_one, &arg1, 0x67, arg2_4[0]) & 0xFF;
    if ( !ret ) {
      // arg3: Width, 8 bit mode, or 16 bit mode
      ret = checked_atoi(arg_three, &width, 0x69, 0) & 0xFF;
      if ( width == 8 || width == 16 && !ret ) {
        // Arg4: Strobe on (non zero) or off (0)
        ret = checked_atoi(arg_four, &arg2_4[1], 0x69, 0) & 0xFF;
      } else {
        ret = PARAM_ERROR;
      }
    }
  }
  // Returns a pointer to isp_command[15] = {0x30, 0x00} OR to ascii
  // printed error code (probably)
  ascii = to_ascii(ret, isp_command, 15);
  uart0_puts(ascii);
  if ( !ret ) {
    // Do while we have number of bytes left (arg2)
    while ( arg2_4[0] ) {
      // If arg4 is non zero wait for GPIO_IO0PIN to synchronise (strobe)
      if ( arg2_4[1] )
      {
        while ( !(GPIO_IO0PIN & 0x80) ); // Wait for PIN7 to be low
        while ( GPIO_IO0PIN & 0x80 );    // Wait for PIN7 to be high
      }
      // Read the GPIO pins as a BYTE and save it to the given pointer in memory
      if ( width == 8 ) { // 8 bit mode (so 8 GPIO pins and 1 byte in memory)
        v5 = arg1;
        *(_BYTE *)arg1 = BYTE1(GPIO_IO0PIN);
        arg1 = v5 + 1;
        v6 = arg2_4[0] - 1;
      } else {            // 16 bit mode (so 16 GPIO pins and 2 bytes in memory)
        v7 = arg1;
        *(_WORD *)arg1 = (unsigned int)GPIO_IO0PIN >> 8;
        arg1 = v7 + 2;
        v6 = arg2_4[0] - 2;
      }
      arg2_4[0] = v6;
    }
  }
}
```

B NXP LPC ISP analyser

```
import idaapi
import idascript
from idaapi import *
from idc import *
from idautils import *

def has_lfcr_op(ea, chr):
  for i in range(0, 4):
    if idc.GetOperandValue(ea, i) == ord(chr):
      return True
  return False

# Observation: all UART communication has to end with \r\n
# Assumption 1: Finding a function that has \r and \n as an operand will have
# something to do with UART communication
# Assumption 2: The function with the most XrefsTo will be uart0_puts as
# uart0_puts is used much more then decode protocol
def find_uart0_puts():
  candidates = {}
  for f in Functions():
    function = get_func(f)
    heads = Heads(function.startEA,function.endEA)
    lf = False
    cr = False
    for head in heads:
      if isCode(GetFlags(head)):
        if not lf and has_lfcr_op(head, '\r'):
          cr = True
        if cr and has_lfcr_op(head, '\n'):
          lf = True
    if lf and cr:
      candidates[len(list(XrefsTo(function.startEA)))] = function
  keys = sorted(candidates, reverse=True)
  if len(keys) >= 1:
    return candidates[keys[0]].startEA
  return None


# Observation: all ISP commands start with a capital letter
# Assumption 1: Finding switch tables with only capital letters as cases
# will probably be the ISP handler
def find_isp_switch_table():
  data = []
  for f in Functions():
    function = get_func(f)
    buf = jumptable_info_t()
    heads = Heads(function.startEA,function.endEA)
    for head in heads:
      if get_switch_info_ex(head) is None:
        continue

      # Variables
      sw = get_switch_info_ex(head)
      cases = []
      defaults = []

      # Get all switches from base address
      sc = calc_switch_cases(sw.jumps, sw)
      for idx in xrange(len(sc.cases)):
        cur_case = sc.cases[idx]
        for cidx in xrange(len(cur_case)):
          if sc.targets[idx] != sw.defjump:
            cases.append((cur_case[cidx], sc.targets[idx]))
          else:
            defaults.append((cur_case[cidx], sc.targets[idx]))
      cases = sorted(cases, key=lambda case: case[0])

      # Figure out if all cases are only capital characters
      if all(chr(case[0]).isupper() for case in cases):
        data.append(cases)
  return data
```

```python
# Observation: all ISP handlers return results through uart0_puts
# Assumption 1: All handlers have only one external reference
# Assumption 2: All handlers have at least one argument and a
# function to sanitize and/or get it
def find_non_trivial_isp_handlers():
  global uart0_puts
  global checked_atoi
  candidates = []
  for f in Functions():
    function = idaapi.get_func(f)
    function_refs = list(XrefsTo(f))

    # We only want functions that are called just once!!
    if len(function_refs) == 1:
      found = False
      for x in [x for x in FuncItems(function.startEA)]:
        for xref in XrefsFrom(x):
          if xref.iscode:
            if xref.to == uart0_puts:
              found = True
        if found: break
      if found:
        candidates.append(function)

  # Assumption 1: The majority will call the sanitizer
  # function one or more times, so if there is a minority that
  # does not call this function, they are not ISP handlers
  # Assumption 2: The first few function call inside these ISP
  # handlers will be sanitizer functions. Go through all ISP
  # handlers and count what the most called function is in the
  # first 3 calls.
  sanitizers = {}
  for c in candidates:
    callNo = 0
    for x in [x for x in FuncItems(c.startEA) if idaapi.is_call_insn(x)]:
      callNo += 1
      for xref in XrefsFrom(x, idaapi.XREF_FAR): # Will be ONE
        if xref.iscode:
          sanitizers[xref.to] = sanitizers[xref.to]+1 if xref.to in sanitizers else 1
      if callNo >= 3:
        break
  if len(sanitizers) <= 0:
    return []
  checked_atoi = sorted(sanitizers, key=sanitizers.get, reverse=True)[0]


  # Remove all functions that do not call sanitizer function
  isp_handlers = []
  for c in candidates:
    found = False
    for x in [x for x in FuncItems(c.startEA) if idaapi.is_call_insn(x)]:
      for xref in XrefsFrom(x, idaapi.XREF_FAR):
        if xref.iscode:
          if xref.to == checked_atoi:
            found = True
    if found:
      isp_handlers.append(c)

  return isp_handlers

# Entry point
uart0_puts = find_uart0_puts()
checked_atoi = None
if uart0_puts == None:
  print "Was unable to find uart0_puts, this is required for further analyses!"
else:
  print "Scanning for an ISP handler switch table ..."
  switches = find_isp_switch_table()
  if len(switches) >= 1:
    for cases in switches:
      for e in cases:
        print "  case %s: goto 0x%x" % (chr(e[0]), e[1])
```

```
else:
  print "  did not find ISP handler switch table!"
print ""


print "Scanning for ISP handlers that have arguments ..."
handlers = find_non_trivial_isp_handlers()
if len(handlers) >= 1:
  for function in handlers:
    print "  handler %s at 0x%08X" % \
      (GetFunctionName(function.startEA), function.startEA)
else:
  print "  did not find any ISP handler, so this one uses a really different" \
    "system, or dump is not loaded correctly!"
print ""
```