Configuring Unconnected Embedded Devices with Smartphones

Peter Barth(✉), Nicholas Linse, and Rüdiger Willenberg

Hochschule Mannheim, Paul-Wittsack-Straße 10, 68163 Mannheim, Germany
p.barth@hs-mannheim.de

Abstract. We configure embedded devices with a smartphone via NFC using an open, platform independent protocol presented in this paper. A textual device specification defines the types of configuration values for a specific device and integrates the device into the configuration system. The specification needs to be provided by the embedded developer. It is translated into a C library that enables configuration value access, as well as *blob* that contains the compressed configuration metadata. A generic smartphone application interprets the metadata and configuration data read via NFC and allows the modification of the values according to the device specification encoded in the metadata. The modified configuration data can be stored, shared or transferred back to the embedded device. None of the configuration steps need an internet connection, which means data is kept private. Combined with the open protocol and the generic app, this ensures that embedded devices will not become obsolete through vendor decisions, as happens frequently with devices dependent on configuration via cloud services. Embedded developers only need to implement raw read and write binary access to an NFC storage device. The generated artifacts allow to transform that data into an easy-to-use data structure. A prototype system using a fully functional tool chain, a generic Android app and a single-board computer simulating an embedded device has been implemented and evaluated.

Keywords: embedded device configuration · NFC · smartphone app

1 Introduction

Increasingly, embedded devices are connected to the cloud to configure them via the web or a proprietary smartphone application as shown in Fig. 1a). For the device manufacturer, this has three advantages. First, the hardware cost and complexity of the embedded device can be reduced, as no expensive input and output hardware such as displays and keys need to be included. Second, the manufacturer benefits from the configuration data that is collected remotely. Third, customers get locked into the ecosystem of the manufacturer, making it harder to switch to other brands. As devices often stay connected, even if that is not necessary, the vendor may collect usage data to gain insights into the customer

J. Taheri et al. (Eds.): MobiCASE 2022, LNICST 495, pp. 3–17, 2023.
https://doi.org/10.1007/978-3-031-31891-7_1

base. However, privacy concerns make both consumers and business customers hesitant to freely provide usage and even configuration data to their suppliers. Why should a lamp know the WiFi-Password (credentials)? Why should a coffee machine vendor know how much (too much) coffee we drink (usage data) and store that in the cloud? Why should they even know that we drink our coffee stronger in the morning (configuration data)? Why should washing machines, power outlets or heating systems need to be connected to the Internet? At least in Europe, the principle of data avoidance and minimization [15, Chapter II, Article 5] is enforced through the General Data Protection Regulation (GDPR) and most embedded devices could run fine without communicating with a cloud server. In addition, personalized data is sensitive according to the GDPR [15, Chapter I] and collecting it may make embedded device vendors subject to severe penalties for violations when processing, distributing or simply storing that data with the hyperscalers, especially if servers are located outside the European Union [19, Chapter 4.1]. All embedded device vendors are keen on reducing hardware costs and many might even give up data collection to avoid potential GDPR-violations altogether, provided they retain the cost savings and may even receive an influx of more data conscious customers. Furthermore, strategy changes or vendors going out of business may make required remote services defunct and thus, otherwise well-working devices become obsolete. Without vendor lock-in, embedded device manufacturers can offer greater sustainability and are more attractive to to buyers with privacy and environmental concerns.

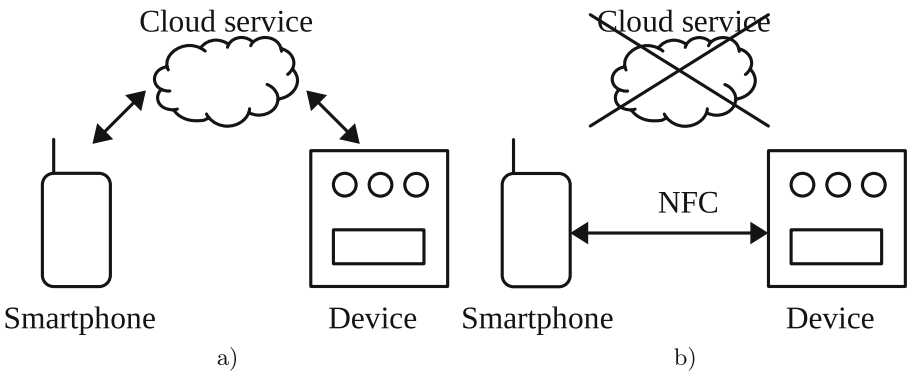


Fig. 1. Configuration of embedded devices via cloud service or direct communication

We propose to replace the omnipresent mid-to-long range wireless chip in embedded devices with an NFC chip [12] and to configure the device with an NFC-capable smartphone and a generic app as shown in Fig. 1b). Thus, the device vendor profits from the reduction in hardware and development cost as well as the easy-to-use configuration process. While there is no need to store data in a cloud, this may still be offered and even easily integrated. But in order to use a device, no registration or other lock-in is necessary. Therefore, devices can

be used and configured beyond their official support lifetime or the lifetime of remote services. For many embedded device vendors, it might even be attractive that no data needs to be stored or processed, and that the customer provides the necessary input/output hardware, the smartphone, themselves. With a generic app, the embedded device vendor has no need to provide a proprietary app, while that still remains an option. Another advantage of remote configuration is that there is almost no configuration logic on the embedded device itself, which simplifies the firmware. To foster adoption of a disconnected approach without remote services, embedded device developers are provided with a tool chain that translates a human-readable representation of the configuration data specification into artifacts that are easily integrated into a firmware without further processing. For embedded devices that is a C struct holding the configuration data. What remains for the embedded developer to implement is the reading and writing of binary data to the NFC chip. The generic app can display and manipulate any data based on the metadata that is generated by the tool chain. The interaction and visualization in the app is dynamically created from the metadata and can be personalized. Data sets can be stored locally for later retrieval, reuse, modification and, optionally, sharing. In Sect. 2, we discuss related work for generic configuration data handling. The possible configuration options and its interaction are collected in Sect. 3. In Sect. 4, we provide an overview of the proposed solution architecture, consisting of the specification of the configuration data in Sect. 5, the tool chain in Sect. 6 and the generic app in Sect. 7. With a test bed using a Raspberry Pi we validate and evaluate the approach in Sect. 8. We conclude in Sect. 9.

2 Related Work

Most apps that rely on NFC are limited to process static information from a tag, such as a URL [3] to be redirected to a Website. A more interesting use is sharing initial pairing information over NFC to establish connections between devices over other communication methods [11]. There are projects which use NFC for general peer-to-peer communication [10], but no widespread applications. Using NFC for device configuration [1,7,16] is mostly proprietary and limited to configuring one specific device. Standardization efforts to offer a common approach to device configuration via NFC are not known to the authors.

Generic configuration of, and interaction with, devices is more widespread in the personal computer space. A good example is the Universal Serial Bus (USB). For classes of devices [18], for example input or audio devices, there are generic drivers that offer limited interaction and configuration of all devices of that class. Specific devices have specific features and consequently require specific configuration, which is available in a device descriptor offered to the host [17, Chapter 5]. For classes of devices, the specific device may choose from a selection of data types and additional attributes and thus follow a similar idea as presented in this paper. Likewise, such an approach can be used to configure Peripheral Component Interconnect (PCI) devices [2,14]. This generic approach

to the configuration of, and interaction with, devices leads to a reduction or elimination of device-specific driver code. A similar approach has also been used to handle communication of Internet of Things (IoT) devices with their backend server generically [9] and to control peripherals of programmable logic controllers over a generic interface to increase program portability [5].

A declarative approach to building user interfaces is typical for app development. Many UI frameworks allow for the declaration of layouts (XML on Android, XAML on .Net) and there are efforts to provide generic collections of UI elements [8] to compose complex interfaces with. Most of these declarations are integrated into the app at compile time. Generating interfaces at run-time through received information is rare. In [4] generating interfaces from a declaration at runtime is discussed to enable device specific adaptations of the interface. Another example would be Facebook's Lite app [6]. This application is intended for devices with low processing power and thus all expensive processing is done in the cloud, which only sends a description of how the final interface should look. The application constructs its interface directly from this description.

3 Configuration of Embedded Devices

A wide range of devices, including coffee machines, time-controlled power outlets, central heating units and a signal generator for use in a laboratory, shall be configurable with the approach presented. All configurations can be reduced to a combination of few data types with additional constraints as well as typical, related interactive widgets. The most-used data types are integral and real numbers, selecting one or more out of many possible options and structured combinations such as date and time. As shown in Table 1, for example, a coffee machine needs a selection for the kind of coffee, the strength, the amount of water and a few additional features. To configure these properties, at least input of integral numbers and selecting 1:n or m:n predefined options is necessary.

Typical interaction patterns for 1:n selection might be some radio buttons for a few options or a dropdown list for more. Some people might prefer to specify the strength with a slider, others might want to input an exact number of grams of powder. For the time-controlled multi power outlet, a setting for the operating time range is needed. In addition, we need to select which of the sockets should be operated, an m:n selection, which could be conveniently selected via check marks. Additionally, not just one but multiple active time ranges and socket selections should be possible. As multiple similar time ranges are possible a way to store multiple data sets of a collection of settings is needed.

The signal generator has settings for the type of signal, frequency, and amplitude. Once again, these settings should exist separately for each output. For the signal type, a 1:n selection is needed, while the other settings can be directly input as numbers, as these might need to be exact. Because frequency or amplitude can only be set in a range that is supported by the device, this must be reflected in the settings, so numbers must support numeric limits for the input values. A generator might only support a range of discrete values, which are not

Table 1. Device examples and used data types

device	feature	data type	interaction
coffee machine	type	Integral number	selection (1:n)
	strength	integral number	number input/slider
	amount	integral number	selection 1:n
	additions	array of m booleans	selection m:n
	time control	date+time+boolean	special
	time control select	boolean	switch
power outlet	on/off	boolean	switch
	multiple profiles	array of complex settings	(repetition)
	profile active	boolean	switch
	time range	date+time	special
	active sockets	array of m booleans	selection m:n
signal generator	on/off	boolean	switch
	output	integral number	selection 1:n
	signal type	integral number	selection 1:n
	frequency	integral number	number input/slider
	amplitude	real number	number input/slider

on a linear scale. In this case, a list of the supported values needs to be defined, which is then used to constrain the input number. Thus, storage of the data is straightforward, while the preferred interaction style may vary depending on device and user preference.

The configuration will be received, stored and read mostly on low-end embedded hardware. Thus, hardware requirements and code complexity should remain low, the language of choice is C. The embedded developer should focus on reading the configuration data from a struct to take action based on the contained values. The used protocol and data structures need to be small, simple and natural for an embedded developer. Thus, no dynamic memory allocation shall be used, as this is often not available on low-end microcontrollers and would limit applicability. The communication protocol should provide very low overhead and thus be binary. Converting between communication data and the storage in a struct should be provided by a generated library. Due to the diverse embedded environment and hardware specialties, not to say bugs, accessing and storing binary data to/from an NFC chip cannot be abstracted away and need to be provided by the embedded developer for this specific NFC tag and board.

For the mobile app, we start with widely available Android smartphones. They have the required performance and interfaces that users are accustomed to. NFC is used for communication as it is increasingly openly available on Android smartphones and cheap as well as easy to deploy on embedded devices. Communication can be established by easy and intuitive touching without the need for complicated pairing processes in related technologies such as classical Bluetooth. The need for physical proximity replaces special security protocols.

However, NFC connections are not suitable for transmitting large amounts of data or to hold a connection for a prolonged time. As we only concentrate on the configuration of devices, not direct interactive control, and the transmitted binary configuration data is small, this does not hinder effective usage.

4 Architecture

Configuring an embedded device over NFC, as depicted in Fig. 2, is triggered by bringing the smartphone into proximity of the device. In one data transfer, the smartphone receives and extracts both data and metadata from the embedded device. Based on the metadata, a user interface is dynamically generated, which allows to change the values according to the specified restrictions found in the metadata. After changing the values, the user may decide to write the changed values back and touches the embedded device again. This again triggers transmission of both data and metadata, but this time back to the device. On the device the transmission is registered, and the newly written values are provided to the firmware. Currently, there is no cryptography involved for authentication, integrity, or secrecy of configurations. Proximity is sufficient to change values. It is advisable, that the firmware developer has to check that the metadata is unchanged, as there is no portable way to only write part of the structure back to the NFC chip. Thus, if the metadata is modified, the embedded developer needs to initiate restoring the metadata from a backup ROM, if it has changed. In addition, the firmware developer has to check that the data itself is sensible, which is helped by generated C code.

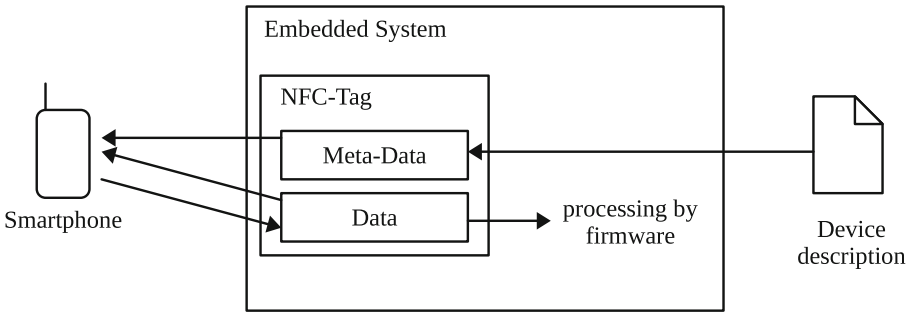


Fig. 2. Configuration of an embedded device with NFC

The components of the system (embedded device, smartphone, and artifacts for firmware development) and their interactions are shown in Fig. 3.

When writing the firmware, the embedded developer specifies the device properties and available settings in a simple YAML-based text document, according to format proposed in Sect. 5. A provided utility detailed in Sect. 6 translates that human-readable description into two binary files consisting of data

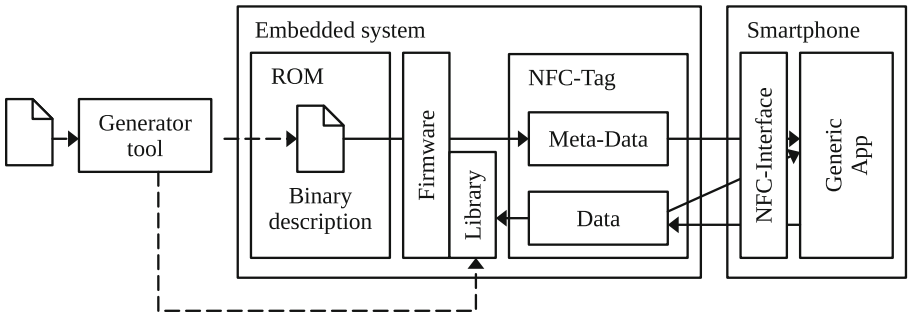


Fig. 3. System architecture for embedded device configuration

with default values and compressed metadata. Both can be combined as a single blob for storage in a backup ROM or as separate blobs for initialization of the NFC chip. Note that the metadata does not need to be understood by the device firmware. In addition, the utility serves as a code generator that generates a C struct and accompanying deserialization and serialization functions, which provide firmware developers with natural access to the stored configuration values. The specification of the device settings is close to existing modern C data types, to simplify their use for embedded developers and to allow easy mapping of settings values to generated struct members. To this end, type sizes are specified explicitly and strings are fixed length and zero terminated. For dates and times, custom structs and types are provided in the generated code, which follow the conventions of the modern C++ date and time library. Furthermore, code to perform sanity checks on the deserialized values is generated and provided as C functions for the firmware developer to use. The embedded developers only need to concern themselves with the specifics of the used hardware, which means implementing functions that read/write binary data from/to the NFC chip and implement reactions to interrupts triggered by the NFC hardware. Typically, the entire information stored on the NFC tag is less than a kilobyte and thus all input/output with the NFC tag is completed in a single operation. Optionally, the metadata can specify that only the data needs to be written back by the smartphone. In this case, the firmware developer has to restore the metadata after every write operation to the NFC tag. However, the benefit is a faster completion of the transfer from smartphone to device, as the data portion is typically small compared to the metadata. This means the smartphone needs to be in contact with the NFC tag for a much shorter time.

5 Device Specification of Configuration Data

To specify how a device is configured, the device developer creates a description of its features and available settings. This description uses the YAML format with a custom extension. An example can be seen in Fig. 4. Here, a time controlled power socket is defined, with four outlets which can be turned on or off during a

```yaml
device_type: Custom_NoTruncate
manufacturer_id: 0xFEFEFE
device_id: 0x010101
firmware_version: 0xFE
protocol_version: 0xFE
device_name: "socket timer"
properties:
  - type: header
    id: "title"
  - type: array
    id: "program"
    max_entries: 4
    properties:
      - type: bool
        id: "program_active"
        dependencies:
          True:
            - "active_range"
            - "active_sockets"
          False:
        default: True
      - type: divider
      - type: time_range
        id: "active_range"
        default: "00:00:00;23:59:59"
      - type: n_out_of_m
        id: "active_sockets"
        entries:
          - "sock1"
          - "sock2"
          - "sock3"
          - "sock4"
translation_data:
  #include "en.yaml"
```

Fig. 4. Device specification for a timed power socket

specified time range. Every specification starts with a block of device information (lines 1–6). It contains unique identifiers for the device and information about its type. This is followed by a list of all available settings (lines 7–32). Each setting has a type and, dependent on the type, zero or more additional properties. A setting of type `bool` for example contains an identifier, the identifiers of settings which depend on its state, and a default initialization value. The type `array` is special, as additional settings can be nested within it. All nested settings will be repeated a given number of times (`max_entries`). This allows storing multiple sets of the same data, while not inflating the metadata description with repeated setting declarations. At the end of the declaration, a list of translations is defined (lines 33–34). Here, the custom addition to the YAML format can be seen in use. An include system, analogous to the C preprocessor include mechanics, can be used to extract parts of a specification into separate files for easier reuse. As YAML is indent-sensitive, the indentation of the `#include` statement is applied

to all included lines when expanding the statement. The include statements are processed by the provided tool chain before parsing the actual YAML content. As the character **#** begins a comment in YAML, the documents are still valid YAML without preprocessing, albeit without the included content.

```yaml
- language: "en"
  translations:
    "title": "Time controlled outlet"
    "program": "Programs"
    "program_active": "Program enabled"
    "active_range": "Active time range"
    "active_sockets": "Active outlets"
    "sock1": "Socket 1"
    "sock2": "Socket 2"
    "sock3": "Socket 3"
    "sock4": "Socket 4"
```

Fig. 5. Translation specification

Translations are used to display the names of settings in different languages in the smartphone application. They are defined as a list of entries, each defining a language and a mapping as pairs of identifiers to translate and their corresponding translations, as seen in Fig. 5.

6 Tool Chain

To help developers with integration of their devices into the system, a custom tool chain is provided. It is implemented in Python and handles code and *blob* generation. It can either be invoked directly from the command line, or imported into other scripts. As seen in Fig. 6, the package takes a device specification as input and generates a corresponding binary blob and C code.

The blob contains the metadata and data in a packed binary format. The metadata is additionally compressed per default, although this can be disabled. This blob is typically not stored directly on the NFC tag. Rather, it should be stored in non-volatile memory on the device, where the embedded system reads the metadata and data blocks and transfers them to the tag during initialization. This allows for an easy factory reset. If the metadata contains translations, developers can optionally specify which languages are packed into a metadata block, which enables regional customization. The blob can then contain several metadata blocks where each contains one of the specified language sets. This is useful, if the NFC tag is too small to hold all languages at once. The device can then load the metadata block into tag memory, which contains only the requested language. To make reading of the generated blob easier for the embedded developer, a table of contents is added to the beginning of the blob. It contains start index, length, and a checksum or hash of each contained block. The checksum can be used to detect changes to or corruption of the blocks on the NFC tag.

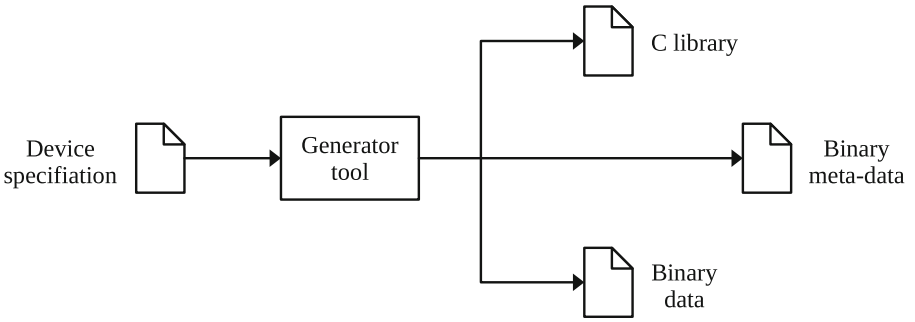


Fig. 6. Artifacts generated by the toolchain

Currently, *MD5*, *SHA1*, *SHA256*, and *CRC32* are available, to allow the developer to choose the most convenient checker available on the embedded platform. In addition to the binary blob, C source code is generated. This code contains a struct definition, conversion functions, and general helper functions. The struct contains members for every setting declared in the device specification. With the generated conversion functions, the values in a binary data block can be transferred into an instance of the aforementioned struct or vice versa. Conversion from struct to binary may be used by the firmware to change configuration values in the NFC storage. Optionally, functions can be generated to perform sanity checks on configuration values and ensure the use of valid values. For an embedded developer, the only steps necessary are creating a device description, generating the blob and code and then implementing writing the blob to an NFC tag, reading the data, and using the generated code to fill a struct with the read data. As a result, the firmware receives a ready-to-use struct, containing only commonly used C data types.

7 Generic Configuration App

We have implemented a reference implementation of the proposed configuration protocol with a generic Android app. This app allows to read a configuration from a device, modify it and then write it back to the device. Additionally, users can save, recall, or share modified configurations. The app supports configuring any device adhering to the aforementioned device descriptions. Note, that this is achieved without needing any kind of device specific code adjustments on the app side. Touching a tag triggers reading the configuration from a device. The data on the tag contains a URL, which is associated with the Android app. This causes launching the app if it is installed, or otherwise opens the URL in a browser, which may offer a download. The application receives the contents of the NFC tag from the Android system and uses a Java library to convert the contained metadata and data into an internal representation of the device configuration. This functionality is extracted into a separate library to ease reuse in other

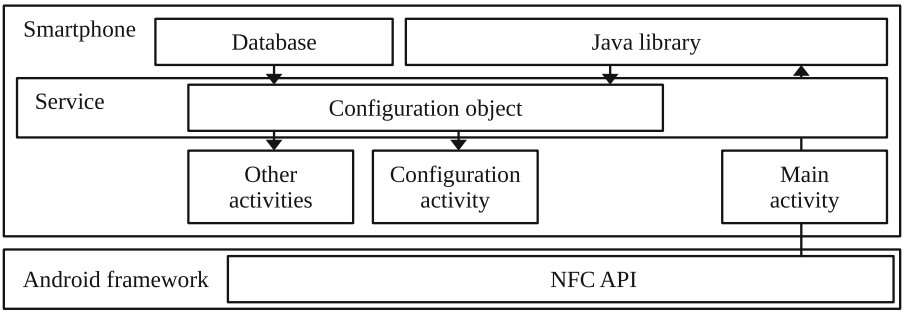


Fig. 7. Architecture of the Android application

applications. This representation is handed to an activity, which constructs a dynamic user interface from it, displaying the available settings and their values and allowing modification. The flow of the tag's data through the application components during this process is shown in Fig. 7.

To create this dynamic interface, we use a library of self-contained widget components, that each handles display and modification of one type of settings. There are some types where a value can be set in different ways. These have multiple interaction components, each implementing a different way to display and input values. As an example, numbers can be input as text or through setting a slider to the desired position. For some settings, such as water temperature, writing a number as text is fine, but for others, such as the brightness of a lamp, a slider might be more comfortable to use. Users may choose their preferred interaction style for a configuration setting per device permanently. The default interaction style is based on heuristics that may change over time and explicitly not embedded into the metadata, as these increase the metadata and only express personal preferences of developers and not necessarily users.

After a user has changed the configuration values, these can be written back to the device. In addition, a copy may be stored in the application for later reuse. This entails, that previously saved configurations can be loaded, changed, deleted, stored and applied. These saved configurations can also be exported as a file in order to share with other users. Importing these files is also possible. In the future, optional sharing over a website can be envisioned.

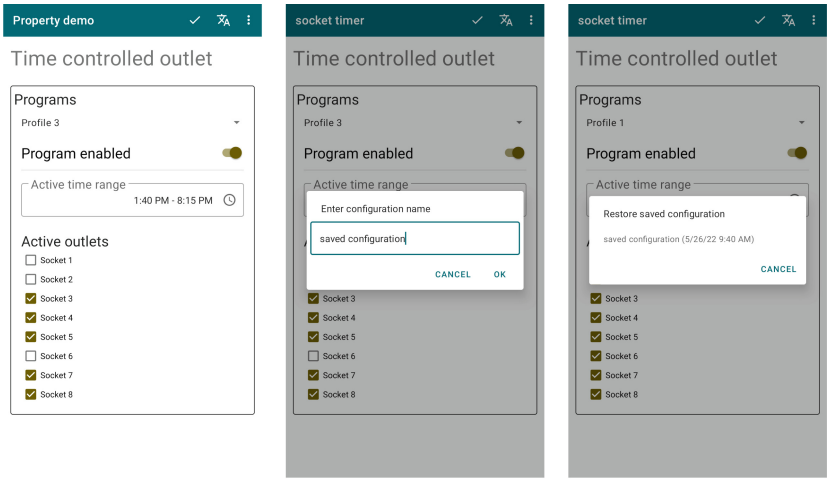
8 Evaluation

A Raspberry Pi is used to simulate different embedded devices. To this end, it is connected to a custom board hosting an NFC tag and several LEDs.

With this setup, a series of fictional devices are simulated. Based on a specification the generated code is tested, and each simulated device is configured by the generic smartphone application. To simulate a device, the generated binary metadata and data is written on the NFC tag. As soon as a smartphone writes

back the configuration data, a command line program reads the data block, converts the data and fills the struct after applying the sanity checks. Finally, the received values are printed on the command line. If applicable, the embodied LEDs are used to signal the state of certain settings.

We detail the development process and simulation of the time controlled switchable power outlet. The values to configure are whether the time control is active, the time range during which sockets are active and which sockets are active. These values are grouped together in a profile and four profiles are available and configurable on the device. First, the developer creates the device specification, as seen in listing 4, from which the binary blob and the library code is generated. For the simulation we use the generic driver to read and write from the development board containing the NFC tag. After touching the tag with the smartphone the app opens, the metadata and data is read and presented to the user as shown in Fig. 8a). The user can modify the setting values, save them in the application as in Fig. 8b), or load previously stored values as in Fig. 8c).



a) b) c)

Fig. 8. Android application user interface for configuration

Accepting modified values triggers a transition to the transmission screen, where the user may request to write the configuration back onto the device. As soon as the smartphone then recognizes the NFC tag again, the changed configuration gets transmitted. Every time a new device is configured, the application stores its identification and associated user customization preferences in a database. These saved preferences can be accessed, modified, or deleted in the application's settings menu, which is shown in Fig. 9a). This is also, where saved configuration values can be managed. In this menu shown in Fig. 9b), saved configurations can be imported, exported or deleted.

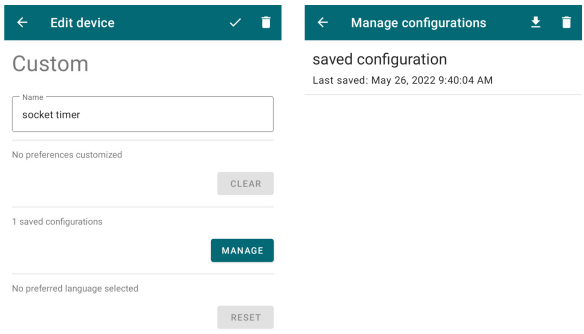


Fig. 9. Android application user interface for management of saved configurations

The presented architecture and implementation is sound and has been used regularly without issues. Binary blob and source code generation is reliable and quick. Because of the generic approach to the smartphone application, a vast range of devices can be supported without further modification. Issues experienced during the realization mostly came down to the used NFC hardware. The NFC tag used in the simulator (NXP NTAG I^2C *plus* [13]) reaches only very low write speeds of approximately 400 bytes per second. Because of this, applying a configuration may take up to two seconds. This means, the smartphone needs to be near the tag for a long time, which makes writing susceptible to transmission errors. To combat this, the data format has been optimized for low memory requirements. Data is packed as efficiently as possible and optional metadata compression leads to a size reduction of up to 60%. Compression rates for different device examples are collected in Table 2.

Table 2. Metadata compression efficiency

Device	Size (bytes)	Compressed (bytes)	Ratio
Demo	1060	592	55,8%
Demo with translations	3451	1324	38,4%
Heater	295	179	60,7%
Heater with translations	888	449	50,6%

The option to only write back data and leave out the metadata shortens transmission time considerably, but requires that the firmware developer restores the tag metadata as well as the data after each configuration. Tag memory is also a scarce resource, with common tags having at most about one kilobyte capacity. With the presented size reduction options, this is usually no hindrance, except for devices with unusually complex configurations. As the size of binary blobs

is fixed, a developer can choose a tag with appropriate capacity. Accidentally or maliciously overwriting tags with invalid data is also possible. This can only be solved by the embedded developer, as tag protection mechanisms are tag specific. However, the generated checksums help with detection of data corruption, while resetting needs to be implemented by the embedded developer.

The Android application works reliably and implementation is straightforward. Minor performance issues are the result of the dynamic creation of the configuration screen. If many widgets are created, which needs to happen in the main thread after the tag is read, delays might become noticeable for the user. However low hundreds of milliseconds were only recorded on a kitchen sink application using all widget types on a low-end smartphone.

9 Conclusion

With the proposed approach we allow easy configuration of hardware-constrained disconnected devices with a smartphone without internet connection. To that end, we can use any suitable NFC chip on the device, a generic app for any NFC-enabled Android smartphone, and a tool chain that translates device configuration specifications to commonly used C data structures for the embedded device developer. Thus, we cater to the needs of privacy-aware consumers and business customers and in addition, increase sustainability of the devices by freeing them from proprietary vendor lock-ins. Obviously, the embedded device vendor has to support the effort. However, by giving up on the necessity (not the option) for cloud connectivity, embedded device developers do not need to run remote services and can save on hardware costs, while still being able to give users an easy-to-use, universally compatible configuration system. The tool chain has been demonstrated to work on lab prototypes using both embedded devices and a Raspberry Pi test bed. The tool chain and the app have been released as open source under https://github.com/ni9l/eput-tools/. Next steps include pilot projects with embedded devices and collecting feedback from the embedded developers. Obviously, newer versions of the app have to support all existing devices configured with earlier tool chain versions. Thus, versioning of the specification format has to be established. When successfully used in more and more embedded devices along with further adoption a standardization process can be envisioned. We hope that this project helps to demonstrate that ease of use, low hardware cost, privacy, and sustainability are attainable simultaneously.

References

1. Abukwaik, H., Groß, C., Aleksy, M.: NFC-based commissioning of adaptive sensing applications for the 5G IIoT. In: Barolli, L., Hellinckx, P., Enokido, T. (eds.) BWCCA 2019. LNNS, vol. 97, pp. 150–161. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-33506-9_14
2. Almeida, N., Alemany, R., Glege, F., da Silva, J., Varela, J.: A software package for the configuration of hardware devices following a generic model. Comput. Phys. Commun. **163**(1), 41–52 (2004)

3. Argueta, D., et al.: Enhancing the restaurant dining experience with an NFC-enabled mobile user interface. In: Memmi, G., Blanke, U. (eds.) MobiCASE 2013. LNICST, vol. 130, pp. 314–321. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-05452-0_29
4. Bisignano, M., Di Modica, G., Tomarchio, O.: An "intent-oriented" approach for multi-device user interface design. In: 20th International Conference on Advanced Information Networking and Applications - Volume 1 (AINA 2006), vol. 2, pp. 186–194 (2006)
5. Eisenmenger, W., Meßmer, J., Wenger, M., Zoitl, A.: Increasing control application reusability through generic device configuration model. In: 2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), pp. 1–8 (2017)
6. How we built facebook lite for every android phone and network. engineering.fb.com/2016/03/09/android/how-we-built-facebook-lite-for-every-android-phone-and-network/. Accessed 21 May 2022
7. Haase, J., Meyer, D., Eckert, M., Klauer, B.: Wireless sensor/actuator device configuration by NFC. In: 2016 IEEE International Conference on Industrial Technology (ICIT), pp. 1336–1340 (2016)
8. Homann, M., Banova, V., Oelbermann, P., Wittges, H., Krcmar, H.: Towards user interface components for dashboard applications on smartphones. In: Memmi, G., Blanke, U. (eds.) MobiCASE 2013. LNICST, vol. 130, pp. 19–32. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-05452-0_2
9. Kim, W., Ko, H., Yun, H., Sung, J., Kim, S., Nam, J.: A generic internet of things (IoT) platform supporting plug-and-play device management based on the semantic web. J. Ambient Intell. Human. Comput. (2019). https://doi.org/10.1007/s12652-019-01464-2. ISSN 1868-5145
10. Lotito, A., Mazzocchi, D.: OPEN-NPP: an open source library to enable P2P over NFC. In: 2012 4th International Workshop on Near Field Communication, pp. 57–62 (2012)
11. Matos, A., Romão, D., Trezentos, P.: Secure hotspot authentication through a near field communication side-channel. In: 2012 IEEE 8th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), pp. 807–814 (2012)
12. NFC Forum: NFC Data Exchange Format (NDEF). NFC Forum, 1.0 edn. (2006)
13. NXP Semiconductors: NTAG I^2C plus: NFC Forum T2T with I^2C interface, password protection and energy harvesting. NXP Semiconductors, 3.5 edn. (2019)
14. Schüpbach, A., Baumann, A., Roscoe, T., Peter, S.: A declarative language approach to device configuration. ACM Trans. Comput. Syst. **30**(1), 1–35 (2012)
15. The European Parliament and the council of the European Union: Regulation (EU) 2016/679 of the European parliament and of the council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/EC (general data protection regulation) (2016). data.europa.eu/eli/reg/2016/679/oj
16. Ulz, T., Pieber, T., Höller, A., Haas, S., Steger, C.: Secured and easy-to-use NFC-based device configuration for the internet of things. IEEE J. Radio Freq. Identif. **1**(1), 75–84 (2017)
17. USB Implementers' Forum: Device Class Definition for Human Interface Devices (HID). USB Implementers' Forum, 1.11 edn. (2001)
18. Defined class codes. www.usb.org/defined-class-codes. Accessed 12 May 2022
19. Škrinjar Vidović, M.: EU data protection reform: challenges for cloud computing. Croatian Yearb. Eur. Law Policy **12**(1), 171–206 (2016). hrcak.srce.hr/174312