SVOX Pico
Core System

Software Architecture and
System Development Guidelines

Version 1.7

April 20, 2009

Contents

1 Introduction

1.1 Document Goals

The SVOX Pico system is a software solution aimed at enabling Text-to-Speech (TTS) functionalities in low CPU/memory platforms. The Pico core system will be deployed as a set of libraries accessible through an API available to application developers.

While benefitting from SVOX knowhow and 'lessons learned', SVOX Pico is an entirely new TTS system designed from scratch and coded in native ANSI-C.

This document describes the Pico core system in terms of the underlying design principles, its architecture and main components, and some guidelines for developers who would like to extend Pico's functionality.

1.2 SVOX Pico Goals

The SVOX Pico project specifically targets small devices with very limited resources. SVOX Pico's design therefore concentrates on these limitations and the targeted applications in mind, fully aware that this has to go at the cost of some of the full flexibility, scalability, and rich variety of features offered by general-purpose TTS system like our SVOX, and also admitting some degradation of TTS quality.

The careful design enables Pico to outperform competing TTS systems in all key features.

1.2.1 Target Applications

- SMS and E-Mail Reader (e.g. abbreviations, date, time)
- Personal Voice Dialler (using Phone book for recognition and confirmation of names)
- Navigation Systems, Verbal turn-by-turn directions
- Animation of avatars (in future versions)

1.2.2 Stated Goals

The main requirements are the following:

- TTS gives control back to calling application after less than 200ms, even on slow CPUs
- Fast response time
- Low latency
- Identical RAM consumption for all languages (around 200 kB)

1.2.3 Minimal Redundancy

The limited resources should be used in an optimized way in favour of quality. Therefore, SVOX Pico should not offer luxury or convenience functionality, e.g.

- no direct audio output
- no built-in command line interface

- no text output (e.g. errors, tracing)
- there is only one way to do one thing (e.g. avoid as soon as possible any API function and text markup doing the same thing)

Functionality needed for building the lingware, running a demo system, testing, etc. should, wherever possible, be implemented as external programs using the SVOX Pico engine via its API. In some cases where this is not feasible, e.g. for tracing, the extra functionality may be opted-in in a special testing configuration (see 'Configurability') but will not exist in other configurations.

1.2.4 Remarks on Lower Flexibility

Pico goals will correspond to losing some flexibility with respect to a standard general purpose system. The Pico system will then be less scalable and allow for:

- one engine (even if conceptually several engines are possible)
- one channel
- one voice
- no (inherent) interrupt control

1.3 Fundamental Design Principles

This section describes the most important design principles for the SVOX Pico core system and the impacts they have on the overall system design.

1.3.1 Platform-Independence

1.3.1.1 Programming Language

The SVOX Pico core system should be created such that the identical source code runs on as many platforms as possible. Therefore source code is written in standard ANSI C, assuming that this is the language most likely supported by the platforms in question.

The use of the standard ANSI-C libraries should be limited to the minimum and made optional considering that on some platforms SVOX Pico will have to run without any access to an underlying library (e.g. platforms without a file system will have to live without physical file IO).

1.3.1.2 Software

Some platforms may not support the needed ANSI-C standard library functions, or the functions may be supported but too inefficient to meet SVOX Pico's resource limitations, while alternative platform-specific functions are better suited. In order to easily port SVOX Pico on all these platforms, basic OS functionality is centralized into an OS abstraction layer. The rest of the system is platform-independent and has no direct access to library functions. Instead, it interacts with the OS abstraction layer via a platform-independent interface.

Some platforms may not support global variables. Therefore, global variables are to be completely avoided.

1.3.1.3 Lingware

Often, lingware sources are available in plain text format and are compiled (and possibly compressed) into platform-independent format (PIL) files that are read by the runtime system and converted to an internal representation. Where loading time is critical, the use of a binary *memory image* (BIN) of the lingware is adopted. BIN files are generally not platform-independent.

For SVOX Pico we define the internal representation such that "PIL=BIN", so that Pico lingware files are platform-independent and at the same time can be loaded at high speed.

For this to work, access to loaded lingware data is done via specialized access functions.

1.3.1.4 Splitting of Text Analysis data and Signal data

For a flexible and efficient management of lingware resources, lingware in Pico is generally split into a lingware resource file containing text analysis (language-dependent) data and lingware resource file containing signal generation (speaker dependent) data.

By this approach different voices can share one common text analysis resource, and installing new or updated resources on a mobile device is more efficient. (E.g. downloading a 'voice pack' or a 'language pack' over the internet

1.3.2 Language-Independence

There is a strict separation between lingware and engine functions, i.e. the engine code should not contain any hard-coded data related to a specific language or voice.

1.3.3 Configurability

Pico should be configurable to allow opting in and out of functionality in such a way, that resources may be kept very small for the base functionality, while additional functionality may be added selectively at the cost of additional resources.

1.3.4 Memory Management

One of SVOX Pico's most stringent limitations is memory consumption. To cope with this requirement, the following design guidelines were set:

- All memory is given by the calling environment using a corresponding startup API function. Memory allocation is managed by the system, within the limits of the memory given at startup.
- Where possible, engine functions should operate on fixed-sized dedicated buffers (e.g. for information passed between processing layers) allocated at engine creation.
- Where inevitable, dynamic memory allocation is managed by the engine within the limits of the memory given at startup. In this case, reaching the memory limits should be anticipated and handled with graceful degradation.

1.3.5 Polling

In many TTS systems, input is done by the application calling a corresponding API function, but output is done via a callback function given to the TTS system at some point. This means that the time until an API function returns is unpredictable and that, once TTS is given enough input to generate a (large) piece of output, it will grab all the CPU it can get to produce this output.

In Pico, a new input doesn’t automatically trigger the processing of this input. Instead, the application has to explicitly provide SVOX Pico with CPU time by calling a “stepping” API function and actively fetching the resulting output. Every API function call has to return within a given time limit (200ms). In order to accomplish a given task, which in general takes longer than this time limit, the stepping function has to be called as many times as necessary. The stepping and fetching of data can be combined into one function, conventionally called “polling”.

1.3.6 Parallel Processing and Multithreading

For the time being, SVOX Pico is required to run only one channel per engine, so that running several channels means running several engines. Since the engine state is encapsulated in an engine “object” (no global variables) and the calling application sets the pace via the polling mechanism, having those engines run in parallel (whether it is genuine multi-threading or pseudo-parallelism) is completely in the hands of the calling application. By keeping lingware resources and processing state (“channel”) cleanly separated, minimal conditions are met to extend SVOX Pico such that multiple channels/engines may share lingware resources.

2 Architecture Overview

2.1 Introduction

This chapter presents the main components of the SVOX Pico core system. The individual components are further discussed in the subsequent chapters.

2.2 Major System Components

The Pico core system can be subdivided into five major layers, with each layer executing different tasks:

a. The **OS interface layer** serves as a narrow interface between the SVOX Pico core system and the operating system. Only modules of this layer call any OS- and library-dependent functions.

b. The **basic functions layer** provides functions that are widely used throughout the system. These functions comprise symbol table management, handling of dynamic arrays, and mathematical operations.

c. The **knowledge management layer** defines internal data structures to hold linguistic knowledge and voice corpora (*knowledge definition* block) and provides functions to load and make knowledge accessible to the actual TTS processing (*knowledge access* block).

d. Th**e run-time processing layer** comprises the definitions of the data structures needed during the actual text-to-speech conversion, the actual TTS processing modules, and a control module that controls the overall TTS conversion chain and which serves as an interface to the SVOX Pico API.

e. The **API and main program layer** comprises the application programming interface of the run-time system and the main programs of the compilation system.

2.3 Modules of the Pico Runtime System

A graphical overview of some modules that make up the Pico Core system Runtime is depicted in Figure 3.1. A full list of all the involved modules (as defined so far) is given in Appendix A.

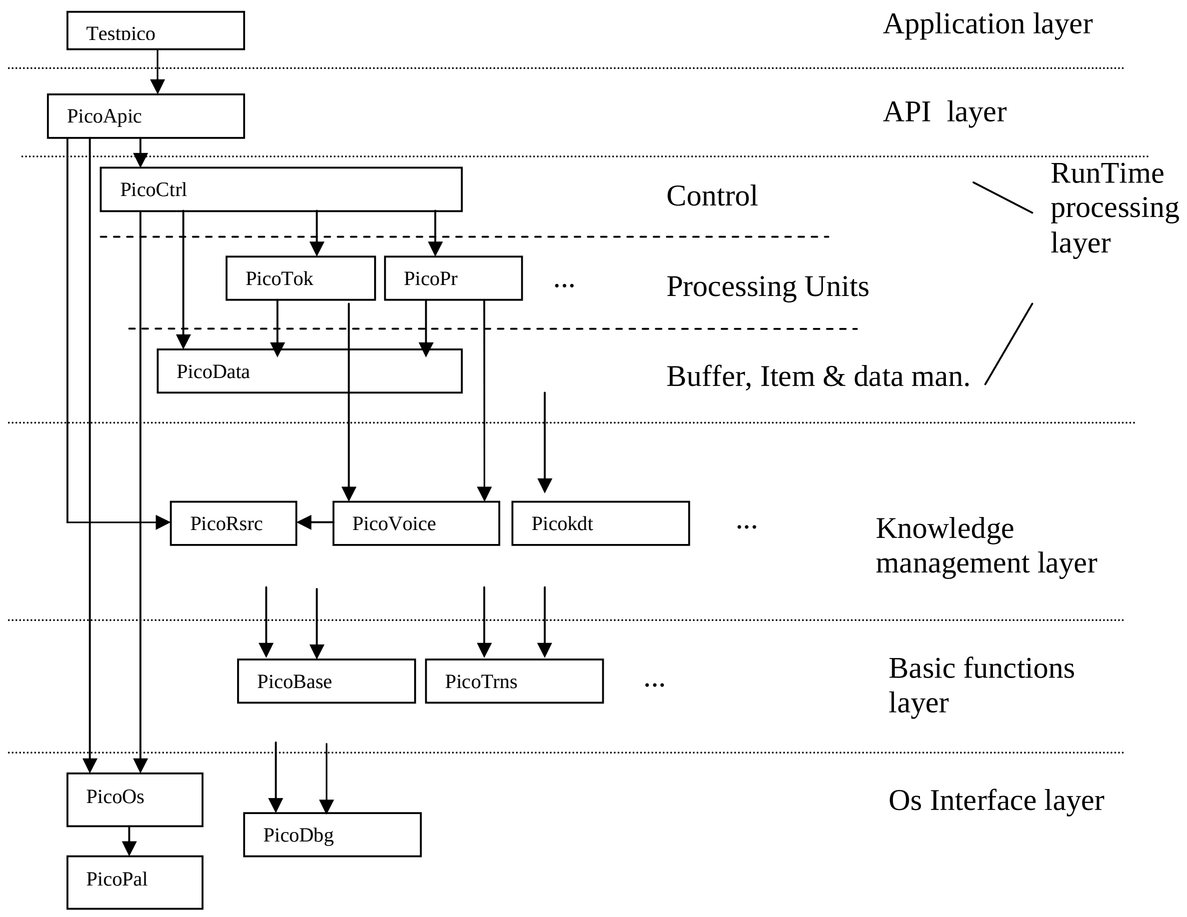


Figure 3.1 Sketch view of some modules of the Pico Runtime core system. The modules in the Applications Layer are deployed to application developers as an example on how to use the API. Blocks with outgoing arrows use functionalities or has dependencies on the blocks pointed to. Cfr. Appendix A for a full list of Pico Modules.

3 Operating System Interface Layer

The operating system interface layer serves as an OS abstraction layer and completely separates the main parts of the SVOX Pico system from any platform-dependent functions.

Some platforms may not support the needed ANSI-C standard library functions, or the functions may be supported but too inefficient to meet SVOX Pico's resource limitations, while alternative platform-specific functions are better suited.

In order to easily port SVOX Pico to all these platforms, basic OS functionality is concentrated into two abstraction layers.

- The first, PicoOS, offers higher-level OS-near functionality, e.g. buffered file access.
- The second layer (PicoPAL) offers an abstraction of the standard library functions.

Both sub-layers may be subdivided (horizontally) such that the run-time system code will only contain those functions that it really uses.

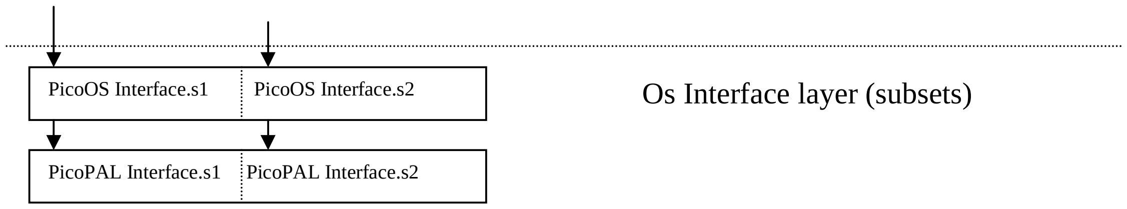


Figure 4.1 Subsets of the PicoOS interface. The deployed runtime could be shipped with different subsets depending on the needed functions.

The implementation of PicoPAL and PicoOS functions may depend on the target platform (e.g. in order to implement library functions that are missing on that platform or to optimize resources consumption).

The OS-dependent implementations are distinguished by simple #ifdef statements, based on a platform symbol which is set for the entire system in one header file (platform.h).

All modules of the SVOX Pico core system which are not themselves part of the OS interface layer may only use OS functions from the PicoOS layer by including the platform-independent header file PicoOS.h.

3.1 Overview

The Pico OS layer functionalities could be better explained if we adopt a terminology and a conceptual partitioning of the full system as follows. The main concepts and ideas are defined in terms of "objects", i.e. in terms of data structures and methods needed to operate on them:

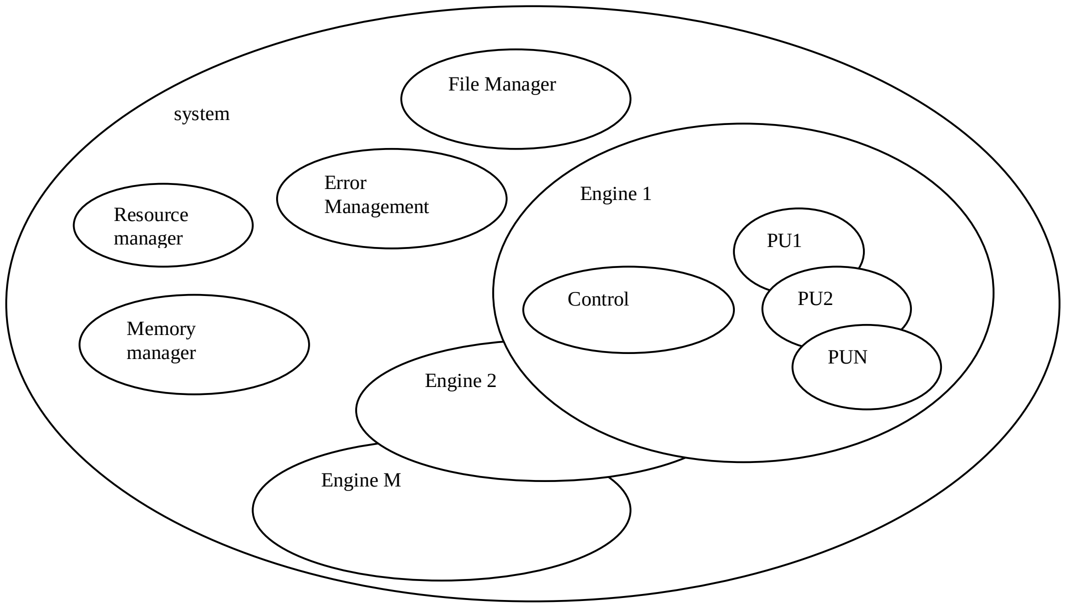
1) **System:** an object that handles things common to all engines, like lingware loading and unloading, creation and deletion of engines, etc. From the application point of view there could be *one and only one System object*, that could make use, internally, of other kind of objects non visible from the outside.

   a) For the system object to be able to manage resource loading and unloading, there should be one and only one **Resource Manager, i.e.** an object that represents the functionalities needed by the TTS engines inside a System object to load and use the linguistic data and signal generation data. *For each System object there could be no more than one Resource Manager*

   b) **Engine**: one out of (possibily) more instances of a TTS device. It includes all features needed to perform TTS conversion, i.e. the runtime API plus the required linguistic and signal generation data. For each engine then, internally, there could be the need of other kind of objects non visible from the outside, for instance:

      (1) **Control Unit:** an object that represents the scheduler of the Engine. For each engine we could have *no more than a single Control Unit*.

      (2) **Processing Units:** one out of more parts of the processing chain. Normally *for each engine we have more than one processing unit*.



Given that description, and with specific attention to what could be "thread sensitive", we could say:

- API functions operate on objects; first parameter to API function is an object handle.
- Two API functions that operate on an identical object must not be called in parallel.
- For each system or engine object, internally, we define a "Common" object
    - It handles common functionalities, including:
        - MemoryManager, governing over memory allocations and deallocations
        - ErrorManager, keeping track of the last error code and text, warnings etc.
        - File manager, keeping track of a list of open files, their respective read/write positions etc.
- This “Common” object should reflect the capability of the Pico runtime to operate engines and system objects as stand-alone entities

This approach is adopted in order to make the Pico Runtime code suitable for application scenarios that could implement some form of multitasking/multiprocessing/multithreading, as far as the TTS service request is concerned.

4 Basic Functions Layer

4.1 Overview

The basic functions layer contains modules that provide basic definitions and functions that are extensively used by the knowledge compiling and the run-time processing part of the SVOX Pico system:

- memory management
- creation and management of dynamically growing arrays
- creation and management of symbol tables
- high-level mathematical and signal processing operations

note: to prevent efficiency losses, no support is given for dynamic memory allocation, All available memory should be allocated and passed at startup by the application using the engine.

5 Knowledge Layer

5.1 Overview

The knowledge layer supports other layers to access, in a consistent and unified way, the data that make up the language-dependent and speaker-dependent content of the Pico runtime system. This data is often called “lingware”.

Lingware data includes text pre-processing and text analysis, signal generation, and potentially other data.

The main entities handled by the Knowledge Layer are the following:

- **KnowledgeBase**: data entity describing one particular aspect of a language or of a speaker characteristics (e.g. lexicon, phoneme property table, pitch prediction tree, etc.).
- **Voice**: essentially a collection of knowledge bases that is required and sufficient to perform TTS in one particular language with one specific set of speaker characteristics.
- **Resource**: essentially a collection of (functionally related) knowledge bases which is a subset of a voice. Lingware is shipped as files each containing a resource, which may be loaded by the system. Usually, the speaker-independent part of the lingware is packaged into one resource (text analysis resource, or TextAna) and the speaker-dependent part into another resource (signal generation resource, or SigGen). Other partitions are possible and may be useful.
- **VoiceDefinition**: a mapping of a voice name to a set of resources that together make up the voice. A voice definition can be created at runtime so that a new voice can be dynamically defined as a combinations of resources.
- **ResourceManager**: entity maintaining lists of loaded resources, voice definitions and voices.

5.2 The unique resource name

Normally Resources are loaded into RAM from resource files. Once a Resource is loaded, it is referred to by its unique resource name (URN) throughout the system. The URN is part of the Resource content and is read when the Resource is loaded. Using the URN, a Resource can be clearly identified without bothering about the paths and file names where the Resource was loaded. The resource name should be globally unique, so that two resources with different content imply that their URN is different.

Using the URN to control the use of resources implies that the URNs must be known to the application, e.g. managed in a database. In some cases the application designer may prefer to use the resource file names to identify resources. In this case, the URN may be retrieved from the loaded resource using an API function.

5.3 Voice and Voice Definition

A voice definition describes a voice as a collection of resources given by their resource names, and assigns this voice a voice name.

“VoiceName”

    URN1,

    URN2,

    .....

The VoiceDefinition may be built at any time before its first usage by calling

createVoiceDefinition(resourceManager, voiceName)

to create an empty voice definition of a given name and one or more of

addResourceToVoiceDefinition(resourceManager, voiceName, resourceName)

to add resources to the voice definition.

A Voice is created using a voice name previously introduced in a voice definition by calling:

createVoice(resourceManager, voiceName, &voice);

While a resource consists of knowledge bases physically loaded into memory, a voice is essentially a list of handles referring to such knowledge bases in one or more resources. Several voices may refer to the same resource.

The resource manager keeps track of all resources, voices, voice definitions and the interdependence between resources and voice definitions. For instance, it will make sure that no more than one copy of a resource is physically loaded, and that resources are not unloaded as long as there are voices referring to them.

5.4 Resources Format Definition

5.4.1 Preliminary Remarks

A resource file contains a **set of knowledge bases** together with a **header** giving some information about the resource and a knowledge base **index** giving information of the position inside the resource and the size of each knowledge base.

Resource files must be platform-independent, but there should be no transformation needed when loading resource files into RAM. This is achieved by assuming the resource to be a **byte sequence**, where the actual meaning of each byte is known and handled by access functions specialized for each particular type of information. From the point of view of the resource manager, knowledge bases inside a resource are just byte sequences, with a given position and size. Numeric information (counts, sizes, offsets) inside resources is encoded as **little-endian unsigned 1-, 2- or 4-byte integers**.

As mentioned earlier, a voice is a (list of handles to the) full set of knowledge bases needed to perform TTS. In order for the resource manager to know which knowledge base in a resource has which role in the voice, “into which slot of the voice it should be filled in”, each knowledge base is assigned a **role identifier**, which is a number between 1 and 255. Each role identifier is associated with an informative **role name**.

5.4.2 Binary Structure of a Resource File

The following is an exact specification of a general resource files as it lists all the contents in the exact order and indicating the allowed sizes.

Each resource file is composed of 5 sections, with each section starting at a 4-byte aligned position.

5.4.2.1 Section A: Foreign header (optional, 4-byte aligned)

(1) The resource may be prepended by (e.g. vendor specific) binary header of size 0-64 bytes, in a multiple of 4 bytes. The foreign header contents may not contain the SVOX Pico header, see B(2). Header.

5.4.2.2 Section B: Pico Header (4-byte aligned)

The Pico header contains a constant string that marks its start plus information about the resource, e.g. date and time of its release, and its name, in form of key/value pairs. These key/value pairs are inserted as clear text so that the contents of a resource file can be verified at a glance without a special tool. The Pico header is composed of

(2) SVOX Pico header: constant 13-byte string (it is actually " (C) SVOX AG ", with each char lowered by 32)

(3) length of the rest of the header (2 byte, excluding length itself)

(4) number of fields (1 byte, number must be <= 10)

(5) a sequence of key/value pairs

- as many pairs as indicated in B(4)
- keys and values must be alphanumeric strings of size 1-32 bytes
- each key and each value is terminated by a space (" ").

The following key/value pairs must be present:

Key	**Possible Value**	**example**	**details**
NAME	Any alphanumeric	de-DE_gl0_sg_1.0.0.0-0-1	should be a unique name for each content
VERSION	any	1.0.0.0-0-1	See SDK Manual
DATE	YYYY-MM-DD	2009-04-01	release date
TIME	HH:MM:SS.xxx	17:01:00.000	release time
CONTENT_TYPE	any	SIGGEN	Usually one of TEXTANA, SIGGEN

(6) filler (0-3 bytes) to make the section size a multiple of 4 bytes.

5.4.2.3 Section C: Length of remaining content

(7) 4-byte number, counting number of remaining bytes in resource (Section D + Section E)

5.4.2.4 Section D: Index of knowledge bases in E

(8) number of knowledge bases in E (1 byte)

(9) sequence of role names of knowledge bases in E (informative)

- each role name is an alphanumeric strings of max 15 chars plus closing space
- as many names as indicated in D(8).

- order corresponds to order of knowledge bases in E

(10) a sequence of kb directory entries

- - as many entries as indicated in D(8).
- - each entry consists of a knowledge base identifier (1 byte), offset (4 bytes) and size (4 bytes)
- - the offset of a knowledge base is the position of the first byte of that knowledge base in E relative to the first byte of C(7).

(11) filler (0-3 bytes) to make the section size a multiple of 4 bytes.

5.4.2.5 Section E: knowledge bases

(12) sequence of knowledge bases (byte arrays), each 4-byte aligned

6 Run-Time Processing Layer

6.1 Overview

6.1.1 Modules of the Run-Time Processing Layer

The SVOX Pico runtime processing layer can be subdivided into 3 sub-layers:

- Processing Units Layer: deals with the elementary steps implementing TTS processing. The elementary steps are defined according to the linguistic view of the TTS process.
- Data Structures and Basic Operations: deals with the definitions of the data structures involved in the processing unit mechanism, and with the operations needed to build, manage and dispose these structrures, for instance buffers, items etc.
- Control Layer : deals with controlling the elementary steps in a way such that responsiveness and efficiency is guaranteed. It also deals with the initial startup phase of the system, and with the input and output from and to the application requesting the TTS service.

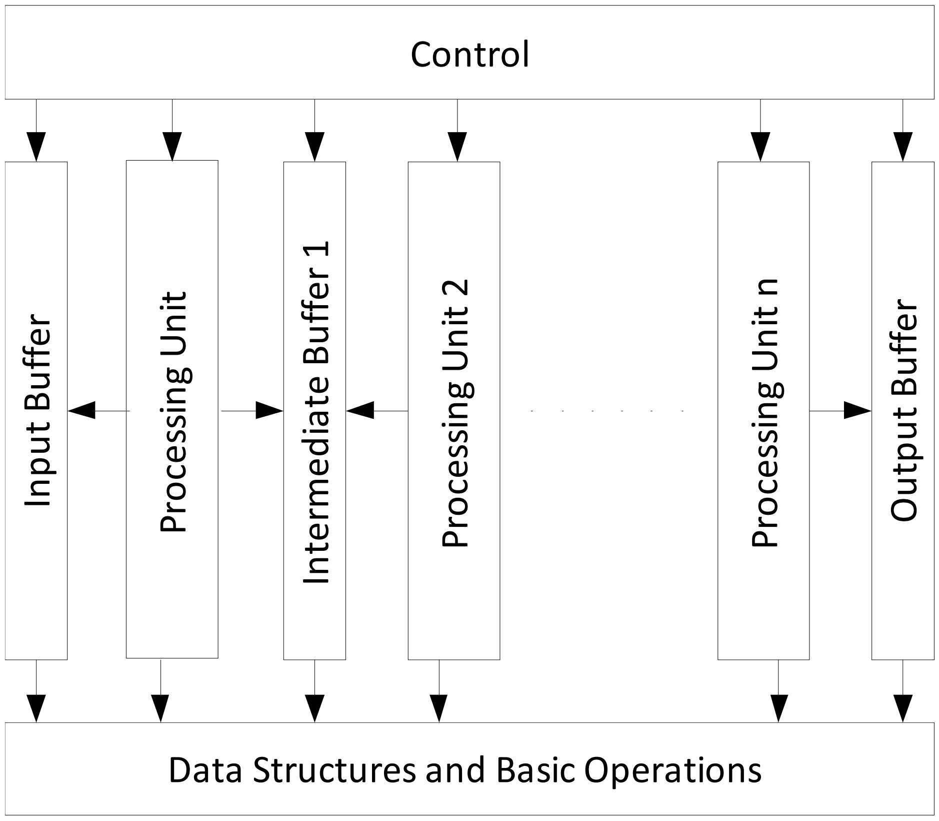


Figure 6.1 Block diagram of the SVOX Pico runtime processing layer.

6.1.2 Basic TTS processing Scheme

TTS processing in the SVOX Pico system is based on a chain of Processing Units (PU), passing intermediate data from one PU of the TTS chain to the following. Each PU transforms its input data in some way, using some of the knowledge bases, and passes the resulting data on to the next PU in the chain. The input data for the first processing unit are single characters and the output data of the last are sampled signal data.

Passing data from one unit to the next is done via dedicated buffers to decouple units from each other. The data passed through the intermediate buffers is represented in terms of ***items*** that can hold data of different nature (see 6.2.1 for the definition of item).

A control unit handles the parallel operation of the processing units, in a way that the TTS requests are served as soon as possible, balancing the load among the different PUs and leaving room for the calling application tasks (i.e. leaving enough CPU power to the application or to the environment, i.e. to perform system functions). This is obtained through a mechanism triggered by the application to check at any time if there are available samples, after a TTS synthesis command has been issued.

The control unit also handles the system start-up phase, the creation, initialization, and disposal of the processing units and the intermediate data buffers, their chaining together. Furthermore it provides all the functions that are visible in the API, in particular the input from and the output to the calling application.

The Data Structures and Basic Operations layer provides data structures commonly used by the processing units and basic operations to manipulate them.

In order to better explain the behaviour of the Control Unit, let's introduce some fundamental concepts about ***items***, processing ***steps***, and processing unit structure.

6.2 Fundamental Data structures

6.2.1 Items

Data is passed from one PU to the next in blocks named ***items***, whose structure is depicted in figure 6.2.

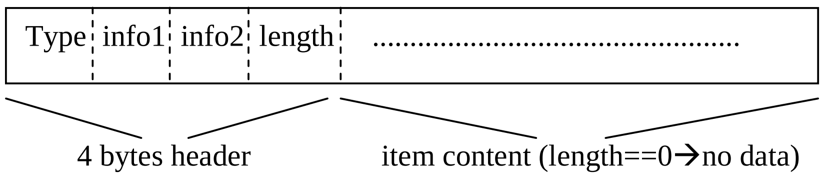


Figure 6.2. The structure of the ***item*** object.

Data types: all bytes are of type picoos_char

A (non exhaustive) list of possible item **types** follows, the full set is defined in picodata.h

• PICODATA_ITEM_WSEQ_GRAPH	a sequence of graphemic words
• PICODATA_ITEM_WORDGRAPH	a graphemic word
• PICODATA_ITEM_WORDPHON	a phonetic word
• PICODATA_ITEM_SYLLPHON	a phonetic syllable
• PICODATA_ITEM_PHONE	a phone
• PICODATA_ITEM_PUNC	a punctuation symbol
• PICODATA_ITEM_CMD	a command
• PICODATA_ITEM_BOUND	a boundary
• PICODATA_ITEM_FRAME_PAR	a parametric frame (before synthesis)
• PICODATA_ITEM_FRAME	a frame (output samples after synthesis)
.....	

The structure of the item is of variable size and content.

For instance, if the item is PICODATA_ITEM_WORD, the field **info1** would bring some information about the POS (values are language-dependent and defined by the lingware).

The item content would bring the representation of the word in graphemic form.

Given that a set of sufficiently large and comprehensive set of types is given, information passing from one PU to the next could be serialized as a sequence of items, whose structure can be reconstructed from the headers.

Then each PU has to deal with an input stream of items that could be managed in several ways. The generic processing of a PU is then:

- Check about the presence of at least one input item.
    - If any
        - Check if the item has to be processed by the PU.
            - If not pass the item to the following PUs.
            - If yes either
                - Consume the input item without producing a corresponding output item (generally buffering it in its local PU storage or altering the state of the local PU storage depending on the content of the input item)
                - Consume the input item and provide items to the following PUs.
                - Interpret the input item as a command and eventually produce one or more output items for the next PUs (i.e. FLUSH).
    - If none
        - return control to the control unit with an IDLE return code

6.2.2 Processing Unit

We use the term “units” to indicate that processing units are not modules. We should rather think of processing units and intermediate buffers as objects (in the OO sense), having an internal state and “methods” assigned to it. One implication is that several processing units can coexist, each with its own state. Also, while the internal functionality might be very different between different processing units, they have a common external behaviour, e.g. their interaction with the control unit.

From the point of view of the Control Unit, the Processing Unit is an OO-like object that has a uniform interface, independent of the specific processing task. The main interface of the PU is represented by

- An initialization function, to do specific PU initialization and memory allocations
- A stepping function to do the processing of input items.
- A termination function to do the cleanup of allocated memory.

In order to conform to the “polling” paradigm mentioned earlier, each processing unit then offers a stepping function, that the Control unit may call repeatedly until the processing unit’s operation is completed.

If we look at the PU form inside, it could be considered as a chunk of code that uses a certain amount of local storage, that gets called from the control unit to do a step, and when input data is available performs the step and eventually produces some output.

The Control Unit itself could be seen as a Processing Unit, in that it has his initialization and termination functions, called at system startup, and has a stepping function called from the application requesting the TTS service.

6.3 Control Unit processing

6.3.1 Overview

The typical scenario is then the following

- TTS startup
  - The application initiates the TTS service, by requesting the creation of a voice engine
  - The Control Unit is initiated.
  - The PUs implementing the TTS processing chain are initiated
  - All the PUs and the Control Unit are idle, and do not consume CPU power unless the application calls the Control Unit step function.
- The application puts some character in the input buffer by calling the appropriate TTS API function. The text is copied into the input buffer in the appropriate format, and nothing is processed so far.
- The application checks sometimes the availability of output data, by calling the appropriate TTS API function. This triggers the Control Unit stepping function.
- The Control Unit takes control of the CPU and
  1. Checks his list of processing units
  2. Determines the PU to be activated and calls his stepping function

3. Updates the status of the called PU and of the next PU to be activated at next call.
4. Returns control to the application, returning also the number of samples available on the output buffer, if any (otherwise 0)

- If there are samples available, the application gets the data calling the appropriate TTS API function and buffers them on his system memory.

The key points are

- Duration of the stepping function for each PU. This should be known in advance, or could be calculated during the initialization phase by the Control Unit.
- A suitable strategy to achieve the good balancing among the time needed for each processing unit to complete a step, the time needed for the whole processing chain to produce an output, and the response time to a single step of the Control Unit towards the application.

Some example could explain better this ***load balancing*** problem. In figure 6.3 the scenario after insertion of some text on the TTS processing chain input buffer is shown.

When the control unit receives control from the application, it has to decide which PU has to be "stepped". The unique PU that could be activated in this scenario is the first one, i.e. the tokenizer, Then the stepping function of it is called: in the simplest case it could perform a stepping function dealing with a single character. This would take a few milliseconds.

After that, the Control Unit returns to the caller.

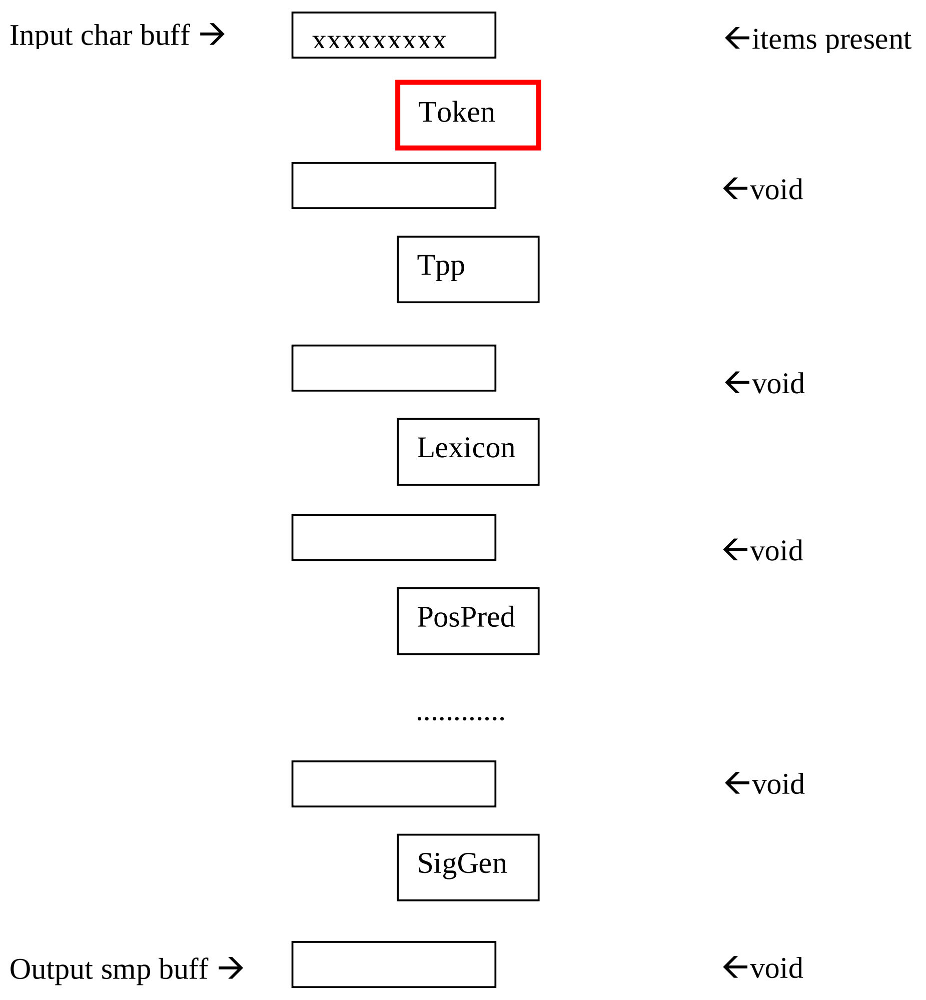


Figure 6.3 Pus and input/output buffers, after text insertion and before first call of the Control Unit stepping function.

After a while the Control Unit (CU) will be called again, and then will have to decide who should be "stepped". Again, it will be the Tokenizer that will continue to buffer his input characters in local storage, unless it is able to separate a token and store it as an item in the output buffer.

At the end of this phase the situation would be similar to what is reported in figure 6.4, where both the input buffers of tokenizer PU and Text Pre Processing PU are partially full.

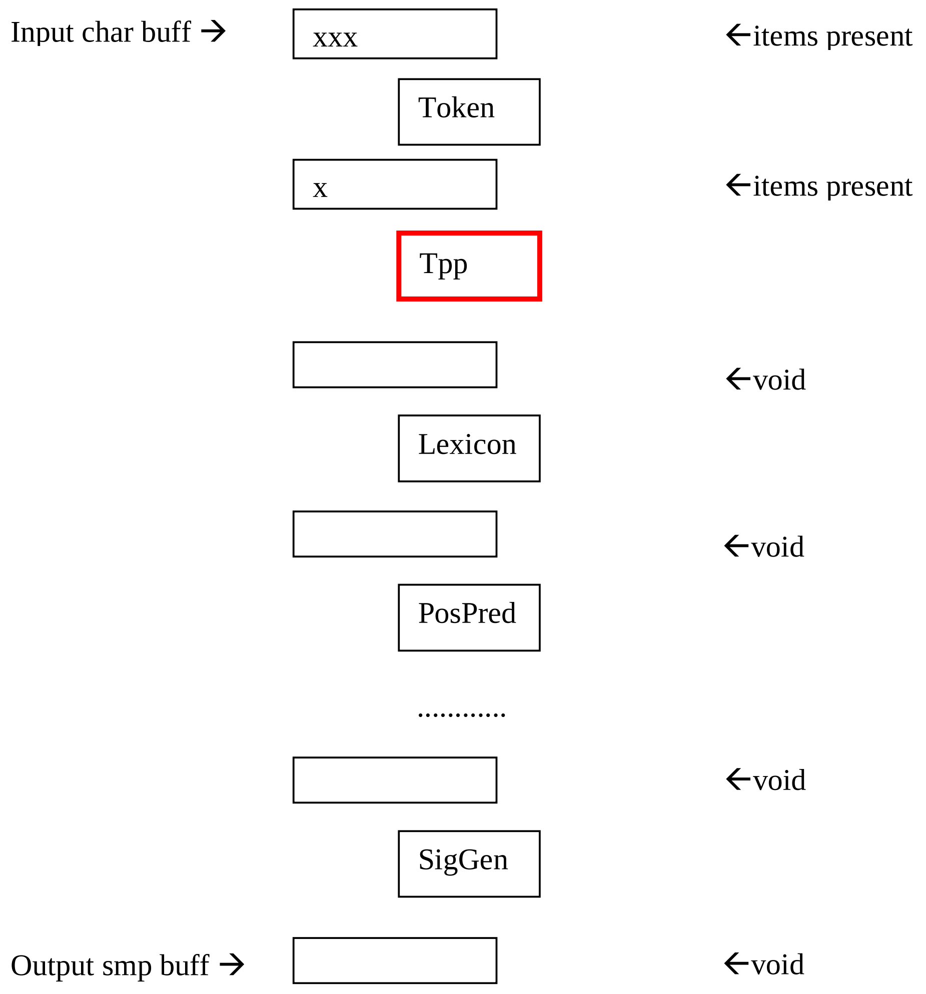


Figure 6.4 Pus and input/output buffers, after text insertion and after some steps of the first PU, i.e. when both the first and the second PU could be triggered.

In this case, when the Control Unit gets called by the application it has to decide who comes first between Tokenizer PU and Text Pre Processing PU. In general priority is given to the ***non idle down-most PU*** on the processing chain.

6.3.2 Control Unit Details and code samples

The current approach is described with some detail below.

6.3.2.1 PU return codes

In order to allow the Control Unit to manage in a simple way the status of each PU, we define the following set of return codes

```
typedef enum picodata_step_result {
      PICODATA_PU_ERROR,   /* this control unit produced an error */
      PICODATA_PU_IDLE,    /* need more input to process internal data */
      PICODATA_PU_BUSY,    /* processing internal data */
      PICODATA_PU_ATOMIC,  /* same as BUSY, but wants to get next time
                              slot i.e. while in an "atomar" operation) */
      PICODATA_PU_OUT_FULL /* can't proceed because output is full.
                              (next time slot to be assigned to pu's
                              output's consumer) */
} picodata_step_result_t;
```

6.3.2.2 PU step function

Then we define an interface function between the Control Unit and the generic PU like this

```
typedef picodata_step_result_t (* picodata_puStep) (
      register picodata_ProcessingUnit this,      /*pointer to PU object*/
      picoos_int16 mode,                          /*activation mode */
      picoos_uint16 * numBytesOutput);            /*# of produced bytes*/
```

This is the function that is invoked for a PU at each step of the corresponding processing unit.

6.3.2.3 PU list in Control Unit

FInally we define, in the local storage of the Control Unit processor, an area where to store information about the active list of current PUs. An example of this follows, where the maximum allowed number of active PU is defined in PICOCTRL_MAX_PROC_UNITS.

```
typedef struct ctrl_subobj
{
    picoos_uint8 numProcUnits;/*actual number of Pus*/
    picoos_uint8 curPU;       /*currently active PU*/
    picodata_ProcessingUnit   procUnit [PICOCTRL_MAX_PROC_UNITS];
    picodata_step_result_t    procStatus [PICOCTRL_MAX_PROC_UNITS];
    picodata_CharBuffer       procCbOut [PICOCTRL_MAX_PROC_UNITS];
} ctrl_subobj_t;
```

Inside this area, we define a status variable `curPU` telling which is the PU to be activated at the next call of the control unit. At system startup `curPU` =0 (first PU).

6.3.2.4 Control Unit scheduling approach

At each request of the calling application the control unit will

- activate the PU corresponding to `curPU` by calling the corresponding step function. This will both return a code indicating the processing status of the PU and `puBytesOutput`, the number of produced bytes (if any).
- Depending on the output of this PU, the `curPU` could change to
  - the following PU (if output items have been produced)

```
 if (puBytesOutput) {
     if (ctrl->curPU < ctrl->numProcUnits-1) {
         /* data was output to internal buffer */
         /* set following pu to busy */
         ctrl->procStatus[ctrl->curPU + 1] = PICODATA_PU_BUSY;
         ctrl->curPU++;
     } else {
         /* data was output to caller output buffer */
         *bytesOutput = puBytesOutput;
     }
     ctrl->curPU++;
}
```

  - the current one = `curPU` (if it has still to consume his inputs:PICODATA_PU_BUSY/ PICODATA_PU_ATOMIC).
  - the previous non idle one = (nothing produced on his outputs and nothing on his inputs: PICODATA_PU_IDLE).

```
while ((ctrl->curPU > 0) &&
      (PICODATA_PU_IDLE == ctrl->procStatus[ctrl->curPU])) {
       ctrl->curPU--;
   }
```

This forces the processing chain to produce outputs as soon as possible. On the other hand different strategies are also possible by customizing the Control Unit.

6.4 TTS Processing Chain

Let's now concentrate on the processing chain content, shown in Figure 6.4.1 .

The order and functionality of the processing units, together with the kind and content of the corresponding lingware, is obviously determined by linguistic considerations, taking into account the required limitations for SVOX Pico.

6.4.1 Some Remarks on the Processing Units

Some PUs make use of general technologies like FST (Finite State Transducers) or DT (Decision Trees).

Decision Tree: the Decision Tree itself is universal (suitable for all instances). There is a trade-off between preparing all data (feature list) potentially needed by the decision tree, with only a small set of simple questions, and a smaller feature list but more complicated questions (essentially calculating features just in time). This is reflected by the implementation.

FST: Sometimes the FST source includes several FST's (offline composition, successive application or online composition)

Lexicon: Because of memory limitations, the lexicon is restricted to function words and frequently used "irregular" words. For the main part of the vocabulary, the part-of-speech, phonemic transcription, morpheme and compound boundaries have to be predicted by appropriate decision trees and tuned by appropriate FSTs.

6.4.2 List of PUs in Pico

Processing units for the Pico prototype implementation are identified first as a list of elementary steps needed to complete the TTS conversion. This list is in Figure 6.4.1, and reports also the technologies that are used for each elementary step. In the product implementation this list could be changed by incorporating one or more elementary steps into one single PU, in order to reduce the number of buffers between PUs. The "incorporation" has to be made on a CPU time consumption base, to be calculated on a "per platform" base, in order to guarantee the < 200 msec requirement for each PU.

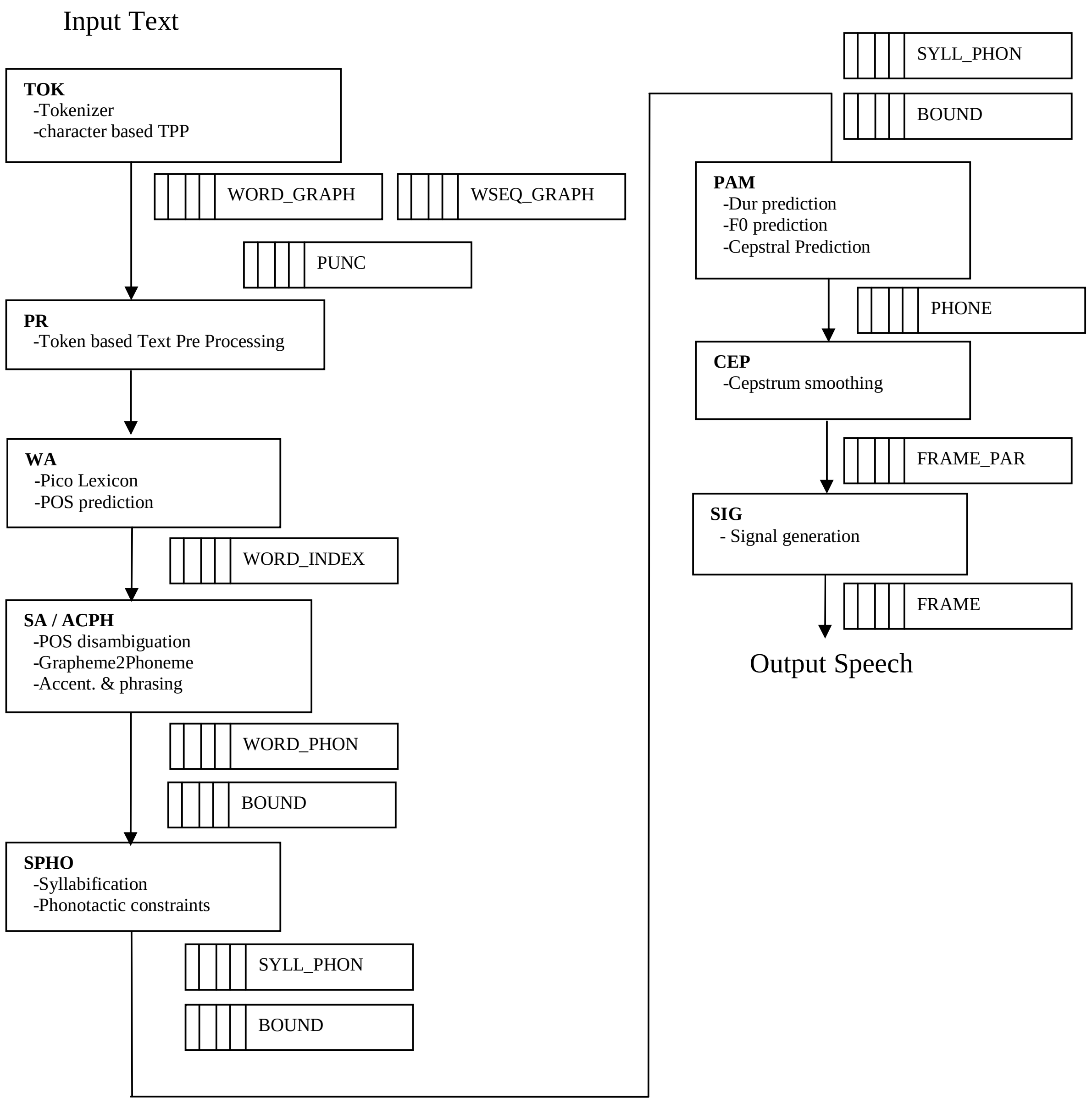


Figure 6.4.1 SVOX Pico Processing Chain represented as a sequence of processing units. Items produced by each PU are listed AFTER the PU itself.

6.5 Processing Unit Interface (Data Transformations)

In this section, the content (not the data structures used in the implementation) of the data passed from one processing unit to the next is described.

How to read:

--	item types that are consumed by the processing layer
+	item types that are produced by the processing layer
=	items that keep their type but their content is modified

Item types not explicitly mentioned are passed without change.

Input:

+ Text (with textual markup); utf8 or utf16 (two separate API functions)

6.5.1 TOK/PR: Tokenizer/Text Preprocessing

Normalized Input:

-- Text

+ Graphemic Tokens containing:

word sequences: sequence of (graphemic) characters (only one of utf8 or utf16); containing word separators if they are known

+ Symbol Tokens containing:

- numbers
- punctuation
- symbols (like "$")

Remark: We will always assume a word sequence in a Token

-> uniform treatment (europ. <-> asian)

-> Later we need word sequence again, anyway

Word separators may be introduced by the Tokenizer/TPP but some or all may be added later. Word separators will actually be POS symbols (assuming up to 50-100 different POS) with "Unknown Word" as the default.

+ Commands

- set speed (absolute value 200-5000; base 1000)
- set pitch (absolute value 200-5000; base 1000)
- set volume (absolute value 200-1200; base 1000)
- set break (type and or pause length)
- wrapper containing Phonetic Token (as if coming from lexicon, ProcUnit B), details to be defined
- set speaker characteristics
- in the future, there will be a need to add additional commands, if additional commands can be added without compromising on higher priority needs they should be added to the system

Other items as ’commands’ (except the ones that has to be managed by this PU), have to be ‘transparently’ passed to next PUs.

6.5.2 WA: Lexicon + POS prediction

= Word Graphemic Tokens, but

- all words separated (POS assigned)
- word sequence tokens contain morpheme, compound boundary and accent symbols

+ Word Phonetic Tokens: words found in lexicon have phones/phonemes instead of graphemes (see next ProcUnit), numbers and symbols are already transcribed here

Other items as “commands” (except the ones that has to be managed by this PU), have to be "transparently" passed to next PUs.

6.5.3 SA/ACPH: POS disambiguation + G2P + Accentuation & Phrasing

-- Word Graphemic Tokens

+ Word Phonetic Tokens:

- Sequence of Phones/Phonemes + Non-Segmental Symbols (POS, Accents, Compound-Boundaries) corresponding word sequences

Other items as ’commands’ (except the ones that has to be managed by this PU), have to be "transparently" passed to next PUs.

6.5.4 SPHO: Transducers for Syllabification and Phonotactic Constraints

-- Word Phonetic Tokens:

+ Syllable Phonetic tokens

= Boundary tokens modified adding syllable boundaries

Other items as ‘commands’ (except the ones that has to be managed by this PU), have to be "transparently" passed to next PUs.

6.5.5 PAM: tree(s) for duration, f0, spectrum (+ tree adapter)

-- Syllable Phonetic Tokens

-- Boundary tokens

+ "Phone Descriptions":

- Phon id
- duration (in number of frames)
- Pitch identifier
- HMM state identifier

Other items as ‘commands’ (except the ones that has to be managed by this PU), have to be "transparently" passed to next PUs.

6.5.6 CEP: Cepstral smoothing and frame transformation

Prepare vectors for final synthesis

-- "Phone Descriptions":

+ "Frames" of parametric data, one item→one frame

- *Phon id*
- Pitch identifier
- HMM state identifier

Other items as ‘commands’ (except the ones that has to be managed by this PU), have to be "transparently" passed to next PUs.

6.5.7 SIG: Synthesis

Generates the sampled data output:

-- "Chunks" of of parametric data, one item→one frame

+ Samples (byte sequence, w/o item structure)

Other items as “commands” (except the ones that has to be managed by this PU), have to be "transparently" passed to next PUs.

There may be PUs following this one, just for debug purposes.

6.5.8 Resume

The following table summerizes the items consumed, produced, modified or passed on at each Processing Unit:

	TOK	PR	WA	SA/ACPH	SPHO	PAM	CEP	SIG
Input text	--							
WSEQ_GRAPH	+	→	--					
WORD_GRAPH	+	→	→	--				
PUNC	+	→	→	--				
WORDINDEX			+	--				
WORDPHON				+	--			
BOUND				+	=	--		
SYLL_PHON					+	--		
PHONE						+	--	
FRAME_PAR							+	--
FRAME								+
CMD	→	→	→	→	→	→	→	→
OTHER	→	→	→	→	→	→	→	→
ERR	→	→	→	→	→	→	→	→

--	consume
+	produce
=	modify
→	Pass or consume

6.6 Implementing a new Processing Unit

In this chapter, we will give the roadmap to insert a new processing unit into the Pico system.

6.6.1 Processing Unit definition

In general, each Processing Unit goes into its own module. The processing unit definition should be made in a corresponding ANSI C header file, i.e. for a new processing XXX unit we should have a "picoXXX.h" file including the following declarations:

```
picodata_ProcessingUnit picoXXX_newXXXUnit(
        picoos_MemoryManager mm,
        picoos_Common common,
        picodata_CharBuffer cbIn,
        picodata_CharBuffer cbOut);
```

6.6.2 Processing Unit implementation

The implementation should coded in "picoXXX.c"

First, we have to define the sub object, i.e. the data that maintains the state of the specialized PU (other than the data common to all Pus). Here is an example that includes a state variable, local input and output buffers with associated pointers, and some useful flags.

```
typedef struct xxx_subobj
{
      /* where to take up work at next processing step */
      picoos_uint8  procState;

      picoos_char   inBuf[IN_XXX_SIZE];   /*internal in buff */
      picoos_uint16 inBufSize;            /*allocated size */
      picoos_uint16 inReadPos;            /*next pos to read from*/
      picoos_uint16 inWritePos;           /*next pos to write to*/

      picoos_char   outBuf[IN_XXX_SIZE];  /*internal output buffer */
      picoos_uint16 outBufSize;           /*allocated size */
      picoos_uint16 outReadPos;           /*next pos to read from*/
      picoos_uint16 outWritePos;          /*next pos to write to*/

      picoos_uint8  needMoreInput,        /*more data needed flag*/

} xxx_subobj_t;
```

We have to implement an initialization method for the sub object

```
pico_status_t xxxInitialize(register picodata_ProcessingUnit this)
{
    xxx_subobj_t * xxx;
    if (NULL == this || NULL == this->subObj) {
        return PICO_ERR_OTHER;
    }
    xxx = (xxx_subobj_t *) this->subObj;
    xxx->inBufSize = IN_XXX_SIZE;
    xxx->outBufSize = IN_XXX_SIZE + Y * PICODATA_ITEM_HEADSIZE + 3;
    xxx->inReadPos = 0;
    xxx->inWritePos = 0;
    xxx->outReadPos = 0;
    xxx->outWritePos = 0;
    xxx->needMoreInput = 0;
    xxx->procState = 0;
    return PICO_OK;
} /* xxxInitialize */
```

Then, we have to implement de-initialization and termination methods for the sub object

```
pico_status_t xxxSubObjDeallocate(register picodata_ProcessingUnit this,
        picoos_MemoryManager mm)
{

    if (NULL != this) {
        picoos_deallocate(this->common->mm, (void *) &this->subObj);
    }
    return PICO_OK;
}


pico_status_t xxxTerminate(register picodata_ProcessingUnit this)
{
    return PICO_OK;
}
```

Then, we have to define a method to create an instance of the PU sub object

```
picodata_ProcessingUnit picoxxx_newXXXUnit(picoos_MemoryManager mm,
        picodata_CharBuffer cbIn, picodata_CharBuffer cbOut)
{
    picodata_ProcessingUnit this =
            picodata_newProcessingUnit(mm, cbIn, cbOut);
    if (this == NULL) {
        return NULL;
    }
    this->initialize    = xxxInitialize;
    this->step          = xxxStep;
    this->terminate     = xxxTerminate;
    this->subDeallocate = xxxSubObjDeallocate;
    this->subObj        = picoos_allocate(mm, sizeof(xxx_subobj_t));
    if (this->subObj == NULL) {
        picoos_deallocate(mm, (void **)&this);
        return NULL;
    }
    xxxInitialize(this);
    return this;
}
```

And finally, we have to insert the real code i.e. the step method

```
picodata_step_result_t xxxStep(
      register picodata_ProcessingUnit this,
      picoos_int16 mode,
      picoos_uint16 * numBytesOutput)
{
    register xxx_subobj_t * xxx;
    picoos_int16 xxxlen;
    picoos_uint8 xxxtype, xxxsubtype;
    if (NULL == this || NULL == this->subObj) {
        return PICODATA_PU_ERROR;
    }
    xxx = (xxx_subobj_t *) this->subObj;

    /* TODO real step code */


    return PICODATA_PU_IDLE;
}
```

6.6.3 Processing Unit insertion on the Control Unit processing chain

6.6.3.1 picodata.h

We should first include the appropriate definitions in "picodata.h". First, we have to specify the processing unit type by adding at:

```
/* different types of processing units */

typedef enum picodata_putype {
    PICODATA_PUTYPE_TEXT, /* text */
    PICODATA_PUTYPE_TOK, /* tokenizer output */
    PICODATA_PUTYPE_ITMW, /* item write output */
    PICODATA_PUTYPE_PREP, /* preprocessor output */
    PICODATA_PUTYPE_DT, /* decision tree output */
    PICODATA_PUTYPE_SYNT /* synthesis output */
    /* …… here it follows the new PU type */
    PICODATA_PUTYPE_XXX /* new processing unit */
    /* etc. */
} picodata_putype_t;
```

Then, we should define the default size of the intermediate buffer by adding it under the following group of defines:

```
/* default buffer size per processing unit  */
#define PICODATA_BUFSIZE_DEFAULT (picoos_uint16) 128
#define PICODATA_BUFSIZE_TEXT    (picoos_uint16)  1 * PICODATA_BUFSIZE_DEFAULT
#define PICODATA_BUFSIZE_TOK     (picoos_uint16)  2 * PICODATA_BUFSIZE_DEFAULT
#define PICODATA_BUFSIZE_ITMW    (picoos_uint16)  2 * PICODATA_BUFSIZE_DEFAULT
#define PICODATA_BUFSIZE_PREP    (picoos_uint16)  2 * PICODATA_BUFSIZE_DEFAULT
#define PICODATA_BUFSIZE_DT      (picoos_uint16)  4 * PICODATA_BUFSIZE_DEFAULT
#define PICODATA_BUFSIZE_SYNT    (picoos_uint16) 16 * PICODATA_BUFSIZE_DEFAULT
/* …… here it follows the new buffer size */
#define PICODATA_BUFSIZE_XXX    (picoos_uint16) xx * PICODATA_BUFSIZE_DEFAULT



Picodata.c
Add PUTYPE handling to get_default_buf_size ???
```

6.6.3.2 picoctrl.c

Check first the definition of ***PICOCTRL_MAX_PROC_UNITS*** in "picoctrl.h" to check against the maximum limit. This value defines internal buffers of the Control Unit list of active PUs and should never be reached.

We should then perform the appropriate modifications in "picoctrl.c".

First of all include the new PU definition by adding the corresponding include file

```
/* processing unit definitions */
#include "picotok.h"
#include "picoitmw.h"
```

```
#include "picoxxx.h"    /*←←←←*/
```

Then, invoke creation method for the newly created PU Type under ctrlAddPU for the appropriate type

```
static pico_status_t ctrlAddPU(
      register picodata_ProcessingUnit this,
      picodata_putype_t puType,
      picoos_int16 last)

   /*...............*/
   switch (puType) {
      /*...............*/
      case PICODATA_PUTYPE_XXX:

        ctrl->procUnit[newPU] =
            picoxxx_newXXXUnit(this->common->mm,
                cbIn, ctrl->procCbOut[newPU]);
        break;
```

Then, we should call this creation method in the picoctrl_newControl

```
   ctrl->numProcUnits = 0;
    if (
            (PICO_OK == ctrlAddPU(this,PICODATA_PUTYPE_TOK,/*last*/0)) &&
            (PICO_OK == ctrlAddPU(this,PICODATA_PUTYPE_XXX,/*last*/0)) &&
            (PICO_OK == ctrlAddPU(this,PICODATA_PUTYPE_YYY,/*last*/1))) {

        return this;
    } else {
        picoctrl_disposeControl(this->mm,&this);
        return NULL;
    }
```

6.6.4 Inside the PU

Once the PU interface is specified, and after its insertion in the processing chain, there is the need to implement the internal of the PU, i.e. its “step” method.

A general, non mandatory, skeleton for a PU, in a conceptual finite state machine representation, could be the one in Figure 6.6.1, in which the states are identified by circles and state transition by arcs, and in figure 6.6.2, in which the PU input, output buffers and local buffers are shown.

This skeleton assumes that for the generic PU:

- There should be (mandatory) a PU input buffer, in which items are written by the previous PU.
- There should be (mandatory) a PU output buffer, in which items are written by the PU itself. There could be (optional) a number of state variables of the PU among which
    - o Current state
    - o input and an output local buffers, managed by the PU itself
    - o Other PU specific state variables

In state 0 (**collect**), we wait for the availability of one or more items in input. If one item is found then the item is copied into a local Input buffer and state changes to 1. The original item in the PU input buffer is removed (actually only pointers are moved), making room for more input items to be produced by the previous PUs.

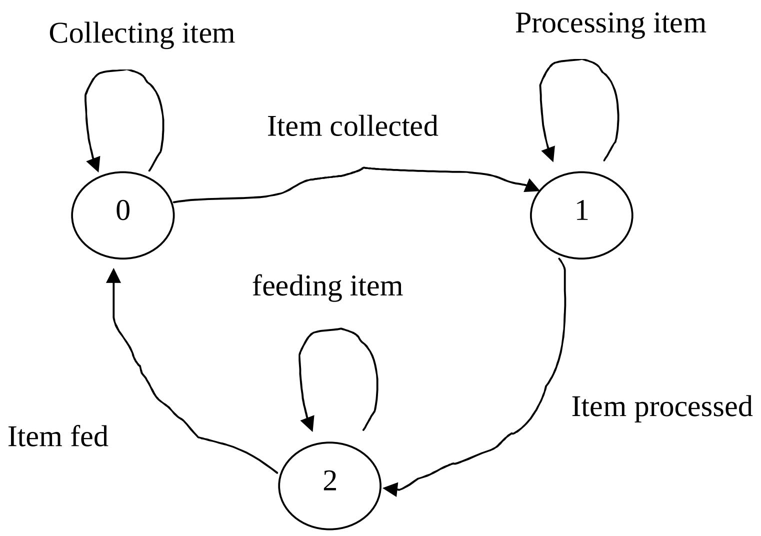


Figure 6.6.1: A finite states representation of the PU “step” processing

Once an item has been identified on the input source, and it is moved in the local PU input buffer, state is moved to 1.

In state 1 (**process**) we elaborate a processing on the item (or items) that are currently on the local input buffer.

We may, at this stage, analyze the currently available items and just copy them in the local output buffer because they cannot be dealth with by this PU, Or we may decide that the number of accumulated items so far are not sufficient to provide an output and as such we may ask for additional input, returning to state 0. Or we may process each item as soon as it arrives, and provide a corresponding output item in the local output buffer. Once some of the input items generate one or more output items, then we could move to state 2, where the actual produced (or copied) items are sent to the output buffer.

In state 2 (**feed**) we have the only task of sending the produced items to the output buffer, managing and updating the appropriate buffer pointers, and then to switch back to state 0 for getting more input.

6.6.4.1 PU return codes

The generic PU could split its processing in several steps and also in sub steps; this is normally achieved by controlling the internal state of the PU and returning to caller even before the completion of item processing. It is important then to understand better what return code to expose to the caller.

The possible return code choices are among the following

- ATOMIC
    - The PU is still processing data and wants to remain scheduled in order to complete the processing.
- BUSY
    - The PU has still items to process but it may yield control to other PU's. The scheduler is free to look for another PU to activate, but will remember that this PU has to be scheduled sooner or later.
- IDLE
    - The PU has no more items to process. The scheduler is free to look for another PU to activate.
- OUT_FULL
    - The PU cannot proceed because the output buffer is full: the scheduler should then activate other PUs in order to consume the items in this PU output buffer.

6.6.4.2 PU returned data vs control unit

The PU step function returns a code, already described in 6.6.4.1, and a number of bytes produced in its output buffer. It is important to consider the combination of the return code and of the amount of data produced, in order to understand the effects of the PU on the overall scheduling strategy.

The control unit will manage the current PU and analyze its return values, i.e. the status and the amount of bytes. The combination of the two will determine the scheduling strategy.

PU returns	**Data produced**	**Control Unit scheduling Action**
ATOMIC	Any	The current PU is not changed
BUSY	>0	The current PU is changed to following one
BUSY	0	The current PU is not changed
IDLE	>0	The current PU is changed to following one
IDLE	0	The current PU is changed to the first preceding

		PU that has a status !=IDLE
OUT_FULL	Any	The current PU is changed to following one
ERROR	Any	The current PU is not changed

6.6.5 Commands for the PU

Among the possible items that a PU may encounter, there are a number that are devoted to support a sort of command interface, that the application can use to control the behavior of the processing chain.

We list in the following some example commands implemented in some of the PU processing stages.

Command	**Notes**
PLAY	Allows PU_ID to get inputs from file with "name.ext"
SAVE	Allows PU_ID to store outputs to file "name.ext"
UNSAVE	Allows PU_ID to stop storing outputs to file "name.ext"
PITCH	Sets the value of baseline pitch
SPEED	Sets the value of baseline speed rate
VOLUME	Sets the value of baseline volume
SPEAKER	Sets the value of speaker modification factor

7 Application Programming Interface Layer

Section 3.3 of the SVOX Pico Manual lists all API functions, giving for each entry a brief explanation of the parameters, input and output, some with additional comments.

Appendix A: Pico Modules

Layer Name	Module Name	Notes
Application Layer	testpico	Any application able to interface the engine via the api
API Layer	picoapi	Application programming interface
Run-Time Processing Layer	picoctrl	Runtime Processing Units Scheduler: takes care of scheduling the suitable processing at each time slice available and to return suitable values to the caller.
	picotok	Tokenizer breaks input text in tokens.
	picopr	Text Pre Processor, partial interpretation of some of the tokens
	picowa	Word analysis, dictionary check for PartOfSpeech assign OR automatic POS prediction
	picosa	Sentence analysis: POS disambiguation, G2P prediction or Lexicon Lookup for G2P, preliminary Syllabification
	picoacph	Accentuation and Phrasing: prominence prediction is carried out and added to words info; boundary prediction is carried out and boundary items between words may be added.
	picospho	Sentence phonology: Phonotactical constraints and final Syllabification. Output items are mainly syllables or boundaries.
	picopam	Phonetic to acoustic mapping: Syllables are mapped to phonemes, phonemes are mapped to sub phonemes, for each sub phoneme spectral, pitch and duration indexes and values are predicted. Output items are mainly phones.
	picocep	Spectral Smoother: Phonemes data are converted to parametric frames according to duration information. Pitch and spectral parametric data is converted from indices to actual values for each frame. A smoothing then is applied to the frame values. Output items are mainly parametric frames..
	picosig	Signal generator: each input smoothed parametric frame is converted into a corresponding output signal buffer. Output items are mainly signal buffers.
Knowledge Layer	picorsrc	Basic resource management

	picoknow	Basic knowledge management
	picokdt	Decision tree knowledge management
	picokfst	FST knowledge management
	picoklex	Lexicon knowledge management
	picokpdf	Prob. Density functions knowledge management
	picokdbg.c	Knowledge management debugging functions
Basic Functions Layer	picobase	common functionalities like string conversion
	picodata	Data management for buffers, items and other
	picotrns	Finite State Transducers management
Operating System Layer	picoos.c	High Level OS-near functionalities
	picopal.c	OS implementation functions (platform specific)
	picodbg.c	Debugging functions