NuTekt NTS-1 digital SDK Source and Template Projects

日本語 (./README_ja.md)

Overview

All source files needed to build custom oscillators and effects for the Nu:Tekt NTS-1 digital kit (https://www.korg.com/products/dj/nts_1) exist under this directory.

Compatibility Notes

Firmware version >= 1.02 is required to run user units built with SDK version 1.1-0.

Overall Structure:

- inc/ (inc/) : Common headers.
- osc/ (osc/) : Custom oscillator project template.
- modfx/ (modfx/) : Custom modulation effect project template.
- delfx/ (delfx/) : Custom delay effect project template.
- revfx/ (revfx/) : Custom reverb effect project template.
- demos/ (demos/) : Demo projects.

Setting up the Development Environment

1. Clone this repository and initialize/update submodules.

```
$ git clone https://github.com/korginc/logue-sdk.git
$ cd logue-sdk
$ git submodule update --init
```

2. Install toolchain: GNU Arm Embedded Toolchain (../../tools/gcc)
3. Install other utilties:
    1. GNU Make (../../tools/make)
    2. Info-ZIP (../../tools/zip)
    3. logue-cli (../../tools/logue-cli) (optional)

Building the Demo Oscillator (Waves)

Waves is a morphing wavetable oscillator that uses the wavetables provided by the custom oscillator API. It is a good example of how to use API functions, declare edit menu parameters and use parameter values of various types. See demos/waves/ (demos/waves/) for code and details.

1. move into the project directory.

```
$ cd logue-sdk/platform/nutekt-digital/demos/waves/
```

2. type `make` to build the project.

```
$ make
Compiler Options
../../../../tools/gcc/gcc-arm-none-eabi-5_4-2016q3/bin/arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -mno-
thumb-interwork -DTHUMB_NO_INTERWORKING -DTHUMB_PRESENT -g -Os -mlittle-endian -mfloat-abi=hard -
mfpu=fpv4-sp-d16 -fsingle-precision-constant -fcheck-new -std=c11 -mstructure-size-boundary=8 -W -Wall -
Wextra -Wa,-alms=./build/lst/ -DSTM32F446xE -DCORTEX_USE_FPU=TRUE -DARM_MATH_CM4 -D__FPU_PRESENT -I. -
I./inc -I./inc/api -I../../inc -I../../inc/dsp -I../../inc/utils -I../../../ext/CMSIS/CMSIS/Include

Compiling _unit.c
Compiling waves.cpp
Linking build/waves.elf
Creating build/waves.hex
Creating build/waves.bin
Creating build/waves.dmp

   text    data     bss     dec     hex filename
   2304       4     144    2452     994 build/waves.elf

Creating build/waves.list
Packaging to ./waves.ntkdigunit

Done
```

3. As the *Packaging...* line indicates, a *.ntkdigunit* file will be generated. This is the final product.

Using *unit* Files

.ntkdigunit files are simple zip files containing the binary payload for the custom oscillator or effect and a metadata file.
They can be loaded onto a NuTekt NTS-1 digital (https://www.korg.com/products/dj/nts_1) using the logue-cli utility (../../tools/logue-cli/) or the Librarian application (https://www.korg.com/products/dj/nts_1/librarian_contents.php).

Creating a New Project

1. Create a copy of a template project for the module you are targetting (osc/modfx/delfx/revfx) and rename it to your convenience.
2. Make sure the PLATFORMDIR variable at the top of the *Makefile* is set to point to the platform/ (../) directory. This path will be used to locate external dependencies and required tools.
3. Set your desired project name in *project.mk*. See the project.mk section for syntax details.
4. Add source files to your new project and edit the *project.mk* file accordingly so that they are found during builds.
5. Edit the *manifest.json* file and set the appropriate metadata for your project. See the manifest.json section for syntax details.

Project Structure

manifest.json

The manifest file consists essentially of a json dictionary like this one:

```
{
    "header" :
    {
        "platform" : "nutekt-digital",
        "module" : "osc",
        "api" : "1.1-0",
        "dev_id" : 0,
        "prg_id" : 0,
        "version" : "1.0-0",
        "name" : "waves",
        "num_param" : 6,
        "params" : [
            ["Wave A",      0,  45,  ""],
            ["Wave B",      0,  43,  ""],
            ["Sub Wave",    0,  15,  ""],
            ["Sub Mix",     0, 100, "%"],
            ["Ring Mix",    0, 100, "%"],
            ["Bit Crush",   0, 100, "%"]
          ]
    }
}
```

- platform (string) : Name of the target platform, should be set to *nutekt-digital*

- module (string) : Keyword for the module targetted, should be one of the following: *osc*, *modfx*, *delfx*, *revfx*
- api (string) : API version used. (format: MAJOR.MINOR-PATCH)
- dev_id (int) : Developer ID, currently unused, set to 0
- prg_id (int) : Program ID, currently unused, can be set for reference.
- version (string) : Program version. (format: MAJOR.MINOR-PATCH)
- name (string) : Program name. (will be displayed on the minilogue xd)
- num_params (int) : Number of parameters in edit menu. Only used for *osc* type projects, should be 0 for custom effects. Oscillators can have up to 6 custom parameters.
- params (array) : Parameter descriptors. Only meaningful for *osc* type projects. Set to an empty array otherwise.

Parameter descriptors are themselves arrays and should contain 4 values:

1. name (string) : Up to about 10 characters can be displayed in the edit menu of the minilogue xd.
2. minimum value (int) : Value should be in -100,100 range.
3. maximum value (int) : Value should be in -100,100 range and greater than minimum value.
4. type (string) : "%" indicates a percentage type, an empty string indicates a typeless value.

In the case of typeless values, the minimum and maximum values should be positive. Also the displayed value will be offset by 1 such that a 0-9 range will be shown as 1-10 to the minilogue xd user.

project.mk

This file is included by the main Makefile to simplify customization.

The following variables are declared:

- PROJECT : Project name. Will be used in the filename of build products.
- UCSRC : C source files.
- UCXXSRC : C++ source files.
- UINCDIR : Header search paths.
- UDEFS : Custom gcc define flags.
- ULIB : Linker library flags.
- ULIBDIR : Linker library search paths.

tests/

The *tests/* directory contains very simple code that illustrates how to write oscillators and effects. They are not meant to do anything useful, nor claim to be optimized. You can refer to these as examples of how to use the different interfaces and test your setup.

Core API

Here's an overview of the core API for custom oscillator/effects.

Oscillators (osc)

Your main implementation file should include *userosc.h* and implement the following functions.

- `void OSC_INIT(uint32_t platform, uint32_t api)`: Called on instantiation of the oscillator. Use this callback to perform any required initializations. See `inc/userprg.h` for possible values of platform and api.

- `void OSC_CYCLE(const user_osc_param_t * const params, int32_t *yn, const uint32_t frames)`: This is where the waveform should be computed. This function is called every time a buffer of *frames* samples is required (1 sample per frame). Samples should be written to the *yn* buffer. Output samples should be formatted in Q31 fixed point representation (https://en.wikipedia.org/wiki/Q_(number_format)).

  Note: Floating-point numbers can be converted to Q31 format using the `f32_to_q31(f)` macro defined in `inc/utils/fixed_math.h`. Also see `inc/userosc.h` for `user_osc_param_t` definition.

  Note: Buffer lengths up to 64 frames should be supported. However you can perform multiple passes on smaller buffers if you prefer. (Optimize for powers of two: 16, 32, 64)

- `void OSC_NOTEON(const user_osc_param_t * const params)`: Called whenever a note on event occurs.

- `void OSC_NOTEOFF(const user_osc_param_t * const params)`: Called whenever a note off event occurs.

- `void OSC_PARAM(uint16_t index, uint16_t value)`: Called whenever the user changes a parameter value.

For more details see the Oscillator Instance API reference (https://korginc.github.io/logue-sdk/ref/minilogue-xd/v1.1-0/html/group__osc__inst.html). Also see the Oscillator Runtime API (https://korginc.github.io/logue-sdk/ref/minilogue-xd/v1.1-0/html/group__osc__api.html), Arithmetic Utilities (https://korginc.github.io/logue-sdk/ref/minilogue-xd/v1.1-0/html/group__utils.html) and Common DSP Utilities (https://korginc.github.io/logue-sdk/ref/minilogue-xd/v1.1-0/html/namespacedsp.html) for useful primitives.

Modulation Effects API (modfx)

Your main implementation file should include `usermodfx.h` and implement the following functions.

- `void MODFX_INIT(uint32_t platform, uint32_t api)`: Called on instantiation of the effect. Use this callback to perform any required initializations. See `inc/userprg.h` for possible values of platform and api.
- `void MODFX_PROCESS(const float *main_xn, float *main_yn, const float *sub_xn, float *sub_yn, uint32_t frames)`: This is where you should process the input samples. This function is called every time a buffer of *frames* samples is required (1 sample per frame). **_xn* buffers denote inputs and **_yn* denote output buffers, where you should write your results.

  Note: There are main_ and sub_ versions of the inputs and outputs in order to support the dual timbre feature of the prologue. On the prologue, both main and sub should be processed the same way in parallel. On other non-multitimbral platforms you can ignore sub_xn and sub_yn

  Note: Buffer lengths up to 64 frames should be supported. However you can perform multiple passes on smaller buffers if you prefer. (Optimize for powers of two: 16, 32, 64)
- `void MODFX_PARAM(uint8_t index, uint32_t value)`: Called whenever the user changes a parameter value.

For more details see the Modulation Effect Instance API reference (https://korginc.github.io/logue-sdk/ref/minilogue-xd/v1.1-0/html/group__modfx__inst.html). Also see the Effects Runtime API (https://korginc.github.io/logue-sdk/ref/minilogue-xd/v1.1-0/html/group__fx__api.html), Arithmetic Utilities (https://korginc.github.io/logue-sdk/ref/minilogue-xd/v1.1-0/html/group__utils.html) and Common DSP Utilities (https://korginc.github.io/logue-sdk/ref/minilogue-xd/v1.1-0/html/namespacedsp.html) for useful primitives.

Delay Effects API (delfx)

Your main implementation file should include `userdelfx.h` and implement the following functions.

- `void DELFX_INIT(uint32_t platform, uint32_t api)`: Called on instantiation of the effect. Use this callback to perform any required initializations. See `inc/userprg.h` for possible values of platform and api.
- `void DELFX_PROCESS(float *xn, uint32_t frames)`: This is where you should process the input samples. This function is called every time a buffer of *frames* samples is required (1 sample per frame). In this case *xn* is both the input and output buffer. Your results should be written in place mixed with the appropriate amount of dry and wet signal (e.g.: set via the shift-depth parameter).

  Note: Buffer lengths up to 64 frames should be supported. However you can perform multiple passes on smaller buffers if you prefer. (Optimize for powers of two: 16, 32, 64)
- `void DELFX_PARAM(uint8_t index, uint32_t value)`: Called whenever the user changes a parameter value.

For more details see the Delay Effect Instance API reference (https://korginc.github.io/logue-sdk/ref/minilogue-xd/v1.1-0/html/group__delfx__inst.html). Also see the Effects Runtime API (https://korginc.github.io/logue-sdk/ref/minilogue-xd/v1.1-0/html/group__fx__api.html), Arithmetic Utilities (https://korginc.github.io/logue-sdk/ref/minilogue-xd/v1.1-0/html/group__utils.html) and Common DSP Utilities (https://korginc.github.io/logue-sdk/ref/minilogue-xd/v1.1-0/html/namespacedsp.html) for useful primitives.

Reverb Effects API (revfx)

Your main implementation file should include `userrevfx.h` and implement the following functions.

- `void REVFX_INIT(uint32_t platform, uint32_t api)`: Called on instantiation of the effect. Use this callback to perform any required initializations. See `inc/userprg.h` for possible values of platform and api.
- `void REVFX_PROCESS(float *xn, uint32_t frames)`: This is where you should process the input samples. This function is called every time a buffer of *frames* samples is required (1 sample per frame). In this case *xn* is both the input and output buffer. Your results should be written in place mixed with the appropriate amount of dry and wet signal (e.g.: set via the shift-depth parameter).

  Note: Buffer lengths up to 64 frames should be supported. However you can perform multiple passes on smaller buffers if you prefer. (Optimize for powers of two: 16, 32, 64)
- `void REVFX_PARAM(uint8_t index, uint32_t value)`: Called whenever the user changes a parameter value.

For more details see the Reverb Effect Instance API reference (https://korginc.github.io/logue-sdk/ref/minilogue-xd/v1.1-0/html/group__revfx__inst.html). Also see the Effects Runtime API (https://korginc.github.io/logue-sdk/ref/minilogue-xd/v1.1-0/html/group__fx__api.html), Arithmetic Utilities (https://korginc.github.io/logue-sdk/ref/minilogue-xd/v1.1-0/html/group__utils.html) and Common DSP Utilities (https://korginc.github.io/logue-sdk/ref/minilogue-xd/v1.1-0/html/namespacedsp.html) for useful primitives.

Web Assembly Builds *(experimental)*

Emscripten builds for oscillators and a Web Audio player are available in the alpha/wasm-builds branch. This is still an experimental feature so some code may not work as expected.

Troubleshooting

Can't compile, missing CMSIS arm_math.h

The CMSIS submodule is likely not initialized or up to date. Make sure to run `git submodule update --init` to initialize and update all submodules.