Graphical Programming of Realtime Applications with LabVIEW for Linux RTAI

Thomas Leibner and G. Schmitz
Science and Engineering Application Datentechnik GmbH
Linder Höhe, D-51147 Köln
Germanythomas.leibner@sea-gmbh.com

Abstract

The programming of real-time applications nowadays involves multitasking using timing demands with microsecond resolution, inter task messaging including semaphores for synchronization of multiple running tasks. For solving this task efficiently a programmer likes to use graphical visualization methods for program flow, messaging and priority dependencies. Instead of using graphical methods only for debugging and schematic overviews and the analysis of problems typical for real-time and multitasking, we show here the use of a graphical programming environment for producing the whole code related to a hard realtime programming example.

The programming environment LabView (tm) by National Instruments provides state-of-the-art tools for solving programming tasks by just doing graphical formulation of algorithms and data flow. One major disadvantage of LabView, not being able to be used as programming environment for hard real-time in Linux kernel space, could be eliminated using the user space API LXRT of the RTAI 'Real Time Application Interface' for Linux.

It is shown, that by implementing a generalized call interface to the LXRT subsystem of RTAI, the user space API of RTAI could be integrated into LabView for Linux. By this extension it is now possible to perform with hard real-time using graphical user space programming. The visual data flow programming of LabView easily visualizes timing and synchronization dependencies.

Although the full LXRT API hasn't been implemented into LabView yet, a basic set of timing and data acquisition functions for hard real-time are shown to be working. A jitter as low as five microseconds on a Pentium III/500MHz makes it possible to have 20 kHz loops with sophisticated computing, e.g. PID closed loop control and audio filtering, by just doing graphical programming. This is another example of the power of the LXRT user space API of the Real Time Application Interface for Linux, giving us the possibility to do hard real-time from user space.

1 Introduction

This talk's intention is to explain the mechanisms behind an example program, which can be found inside the labview section of the RTAI CVS tree[1]. The program is build of two tasks:

1. A feeder task reads sound data from a file and puts it into a realtime communication FIFO. Also, a power spectrum of the chunk of data is computed and graphically displayed. This task runs in normal linux user space context without realtime capabilities.

2. A player task reads sound data from the communication FIFO, decodes them to 1-bit sound and toggles the I/O bit connected to the speaker. This player task also runs in linux user space, but switches itself to hard realtime mode utilizing the RTAI/LXRT realtime API.

Further, the feeder task starts the player task and listens to a semaphore giving a 'ready' sign from the player task. On the other hand, the player task checks a semaphore signalling 'stop' by the feeder task. In case of an empty sound data FIFO (when feeder task ist delayed) the player task toggles the speaker bit, producing an 8kHz beep (useful for scope timing measurements)

2 Dataflow Programming

The program is completely build using a language called *G*, or better known as LabView. Programming in *G* means to depict the dataflow of an application by graphically connecting data sources with data sinks. The connections are done by drawing wires. The color of the wire shows the data type, while its thickness represents the dimension (scalar, array of dimension 1, 2,...) of the data virtually traveling along the wire. Subroutines are represented by building blocks with data inputs (equal to data sinks) and data outputs (equal to data sources). User inputs (e.g. button, value, string) are considered to be data sources and outputs to the user are used similar to data sinks (from the program's point of view). While the wiring of data sources and sinks is done on a sheet called *block diagram* or *back panel*, the 'connection' to the user with buttons, numerical, graphical, and string elements (input and output) is done via the so called *front panel*. The set of a front panel and a block diagram is named *virtual instrument*, or shortened: *VI*. Each VI can be used as a subroutine as well as it can be run (and visually debugged) stand alone. For passing parameters to a VI being used as a subroutine, data elements of the front panel representing parameters and return values are marked and then appear as connector pads on the VI's iconified representation. These iconified versions of the VI will then be used (wired) inside the block diagram of a calling VI.

3 LabView Sound Example

This article, although planned to be a full text explanation of the complete solution, will serve as a handout for the talk beeing held at the RTWS 2002, Boston. Come, listen to the talk and ask questions to gain full understanding of what is going on inside the example. Those with dataflow programming experience (so called *LabViewers*) will wonder how to do microsecond timing from within LabView. Have a look at the back panel of the hard realtime player task. You will find sub-VIs called 'Make Period' and 'Wait Period'. These adjust the timing to the desired 8kHz period and perform a 'wait until next period' inside the loop which reads the communication FIFO and controls the speaker bit.

References

[1] RTAI CVS web interface: http://cvs.rtai.org

List of Figures

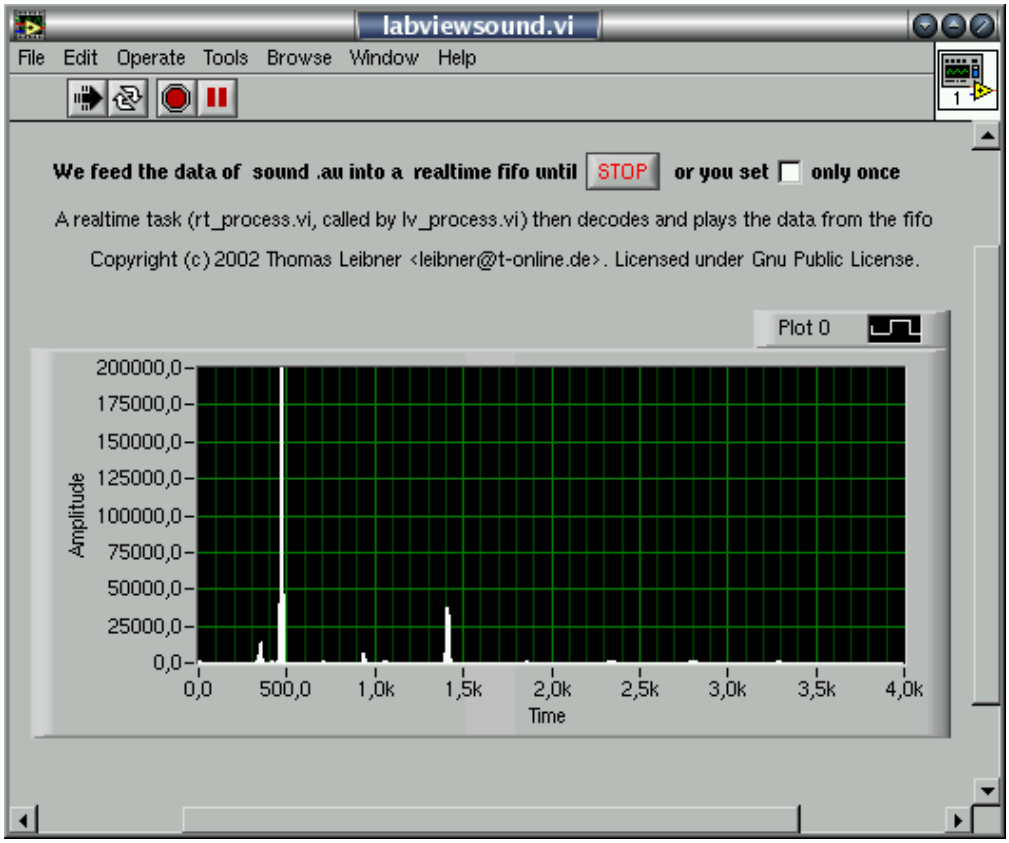

Figure 1: LabVIEW realtime sound playing example. Normal priority user space *.au file playback with simultaneously spectral analysis (base drum & sythesizer rhythm of 'Axel F.')

Figure 2: Hard realtime sound player task with passed parameters: FIFO and semaphore references & error message structures

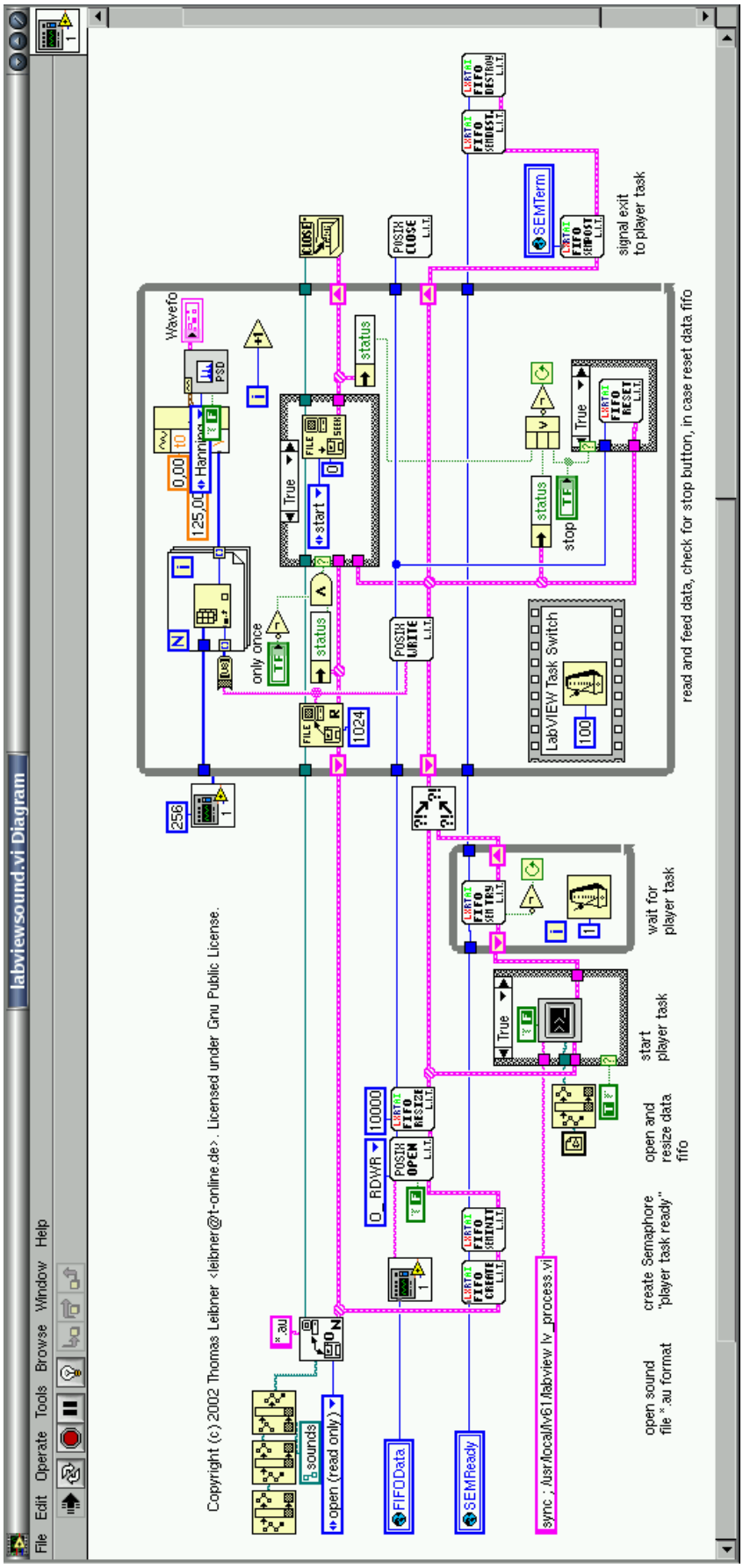

Figure 3: Dataflow coding of the non-realtime LabVIEW sound playing example

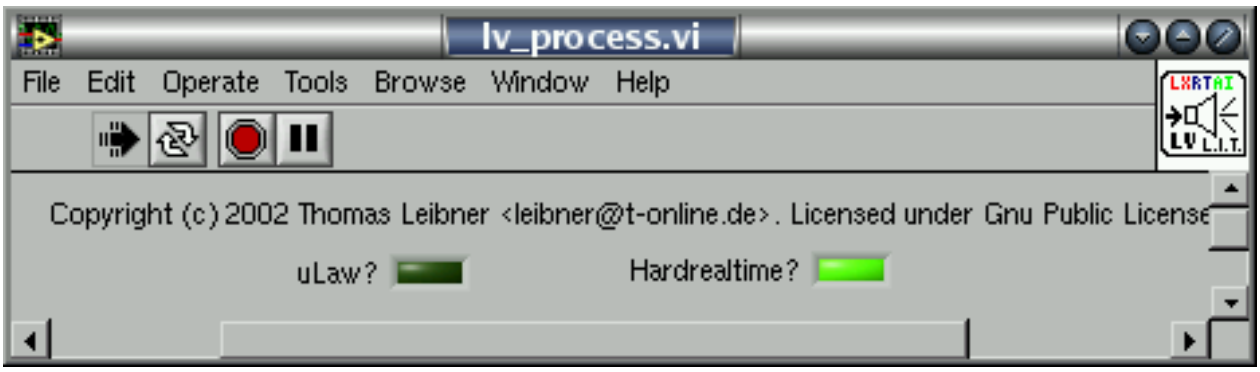

Figure 4: Non-realtime wrapper task for initialization of the hard realtime sound player task

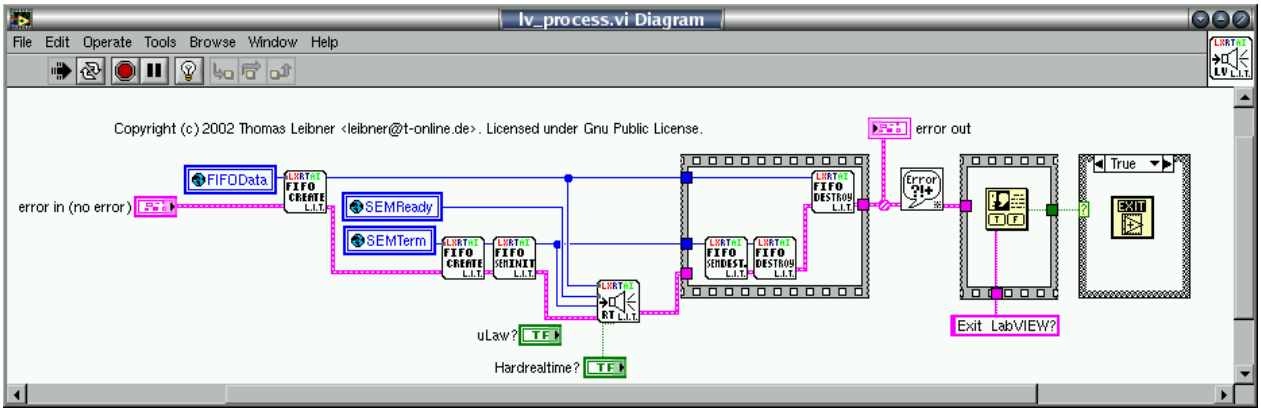

Figure 5: Non-realtime wrapper task: Dataflow coding

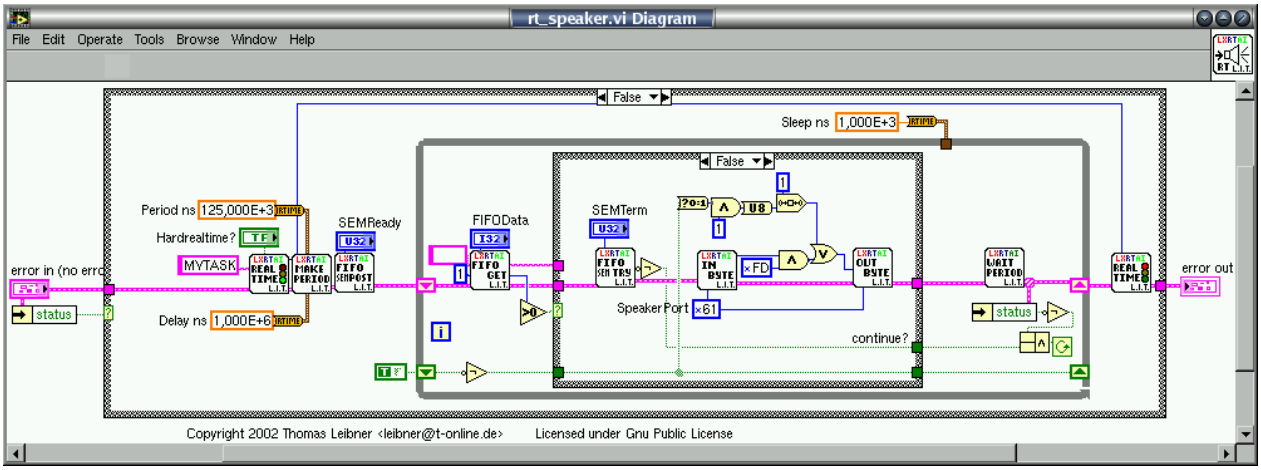

Figure 6: Dataflow coding of hard realtime player task. Shown is case for empty communication FIFO with semaphore checking

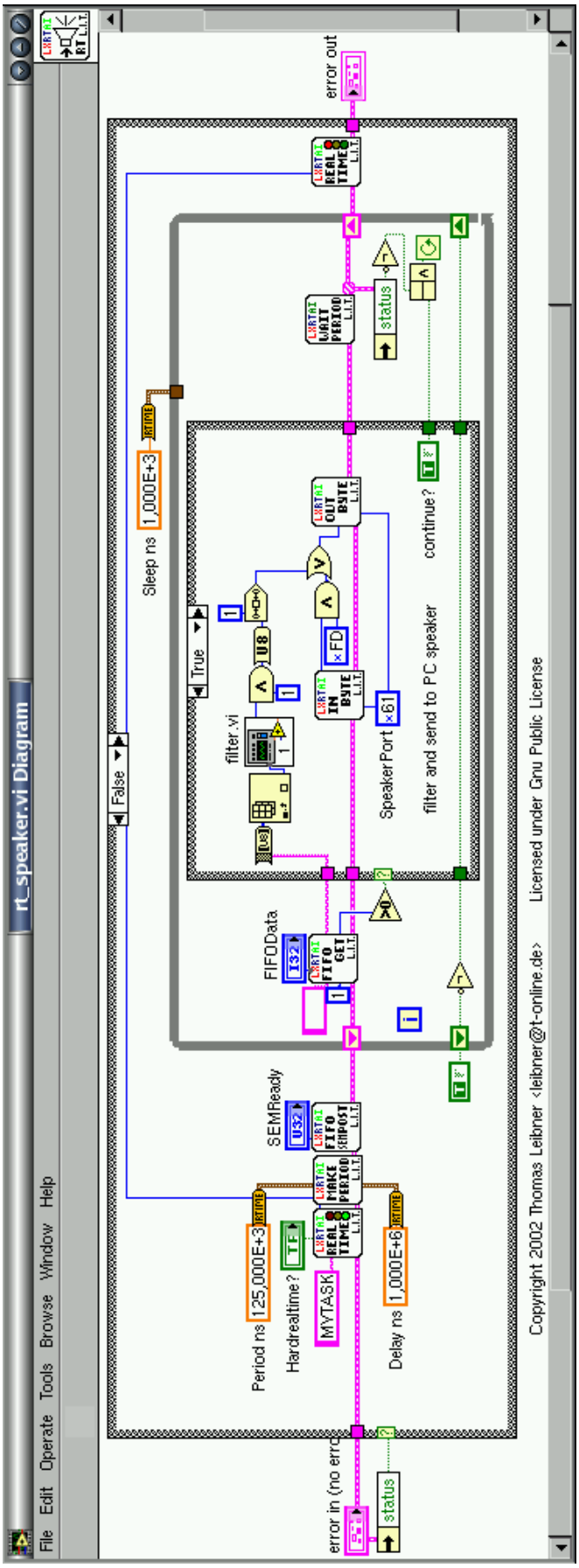

Figure 7: Dataflow coding of hard realtime player task. Shown is case for pending FIFO data