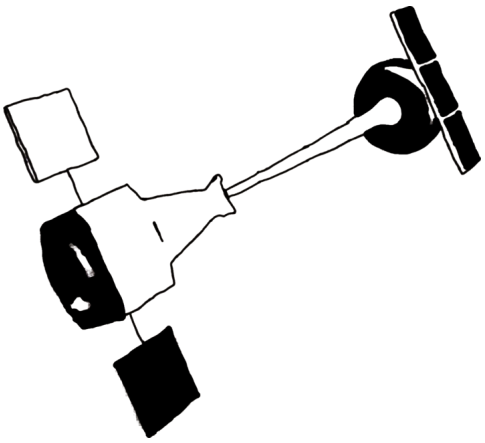
Development of Data Acquisition and Detector Controller Electronics for the Low Energy X-Ray Detector of the Simbol-X Space Mission



Diplomarbeit

eingereicht von

Henry Sebastian Grasshorn Gebhardt

Eberhard Karls Universität Tübingen
Fakultät für Mathematik und Physik
Kepler Center for Astro and Particle Physics
Institut für Astronomie und Astrophysik
Abteilung Astronomie

November 2009

Contents

Deutsche Zusammenfassung

Ziel dieser Diplomarbeit ist der Entwurf und Test eines SpaceWire-Interface-Bausteins für die Simbol-X Satellitenmission. Simbol-X wird eine hohe spektrale und räumliche Auflösung im Energiebereich von 0.5–80 keV besitzen und damit eine wichtige spektrale Beobachtungslücke zwischen bisherigen Missionen wie XMM-Newton und Integral schließen. Dadurch wird es möglich sein Kenntnisse über Beschleunigungsmechanismen in den energiereichsten Systemen des Universums zu gewinnen, wie zum Beispiel der Entstehung von Jets aus Aktiven Galaktischen Kernen. Auch wird solch eine Mission zum besseren Verständnis von Mikroquasaren mit einem Schwarzen Loch oder Neutronenstern im Zentrum einer Akkretionsscheibe führen. Mit einiger Wahrscheinlichkeit wird es außerdem gelingen den Ursprung des Kosmischen Röntgenhintergrundes zu verstehen, da mit Simbol-X zum ersten Mal ein Instrument mit einer hohen Ortsauflösung im harten Röntgenbereich zur Verfügung stehen wird. Einige Quellen und Emissionsmechanismen sind in Kapitel 1 beschrieben, die Simbol-X Mission in Kapitel 2.

Simbol-X umfasst zwei Satelliten, einen mit einer Wolter Optik und einen mit dem Detektorsystem. Diese fliegen in Formation im Anstand von 20 m und fokussieren so die Röntgenstrahlen auf die Detektoren in der Fokalebene. Das Detektorsystem von Simbol-X besteht aus einem Hochenergiedetektor und einem Niederenergiedetektor. Die hier vorliegende Arbeit hat zum Ziel, den Niederenergiedetektor mit einer *SpaceWire* Schnittstelle zu versehen um die Integration mit den übrigen elektronischen Komponenten des Satelliten zu ermöglichen. Der sehr modulare und leicht erweiterbare Entwurf ist nun als VHDL Code in einem FPGA realisiert und wird in Kapitel 3 genauer beschrieben.

Das *SpaceWire* Protokoll ist ein sich etablierender Kommunikationsstandard. Er wurde speziell entwickelt zur Benutzung in Satellitenmissionen und wird auch schon in zum Beispiel dem Herschel oder SWIFT Satelliten verwendet. Die hier dargelegte Arbeit beschreibt und testet das *SpaceWire* Protokoll, das hier in Tübingen zum ersten Mal Verwendung findet, im Zusammenhang mit der Entwicklung der Simbol-X Mission. Das *SpaceWire* Protokoll wird in Kapitel 4 erläutert.

Um die hier entwickelte Schnittstellenkomponente von Simbol-X besser testen zu können, wurde ein spezieller SpaceWire-zu-USB Umwandler entwickelt. Der *Universal Serial Bus* (USB) wurde wegen seiner allgemeinen Verfügbarkeit und einfachen Handhabung gewählt, auch wenn die Details der Implementierung durch diese Wahl etwas komplexer wurden. Jedoch hat der SpaceWire-zu-USB Umwandler den zusätzlichen Vorteil eine generische Methode zu bieten, um große Datenmengen zwischen Software und Hardware auszutauschen. Der Konverter ist in Kapitel 5 beschrieben, und eine knappe aber detaillierte Zusammenfassung des USB Standards ist im Anhang A gegeben.

Leider wurde die Phase B von Simbol-X aus finanziellen Gründen abgesagt. Dennoch findet die hier vorgelegte Arbeit Verwendung insbesondere in Hinsicht auf das *International X-ray Observatory* (IXO), wo das *High Timing Resolution Spectrometer* (HTRS) das SpaceWire Protokoll verwenden und hier am Institut für Astronomie und Astrophysik Tübingen mit entwickelt werden soll.

Preface

> *The letter I have written today is longer than usual because I lacked the time to make it shorter.* — Blaise Pascal

The Simbol-X space mission is a planned space observatory for the 0.5–80 keV X-ray range. Simbol-X will consist of a satellite carrying the optics and another carrying two detectors, a low energy detector and a high energy detector. The two satellites will fly in formation to provide a long focal length of 20 m. Simbol-X would be the first mission to have a good angular resolution and sensitivity in the hard X-ray regime.

The intention is to look at some of the most energetic systems in the universe and further our understanding of the particle acceleration mechanisms active there. Jets are one of the most strange phenomena emanating from the hot center region of accretion discs around black holes and neutron stars in such systems as *Active Galactic Nuclei* and *Microquasars*. X-rays provide a unique view into that high-energy region with temperatures in excess of 10^5 K, where rotationally and gravitationally smeared out iron lines will be much better visible with an observatory such as Simbol-X. Also, the mission is needed to resolve the origin of the *cosmic X-ray background*, to better determine the parameters of recent *supernova remnants* in our galaxy, and to further understand the hot *intracluster gas* between galaxies. The emission mechanisms and sources are the topic of chapter 1.

The present work makes a small contribution towards reaching these goals by providing a *SpaceWire interface controller* for electronically connecting the low energy detector of Simbol-X to the rest of the spacecraft. The detector is a new silicon-based matrix with an extraordinarily high frame rate of 8000 frames per second that will allow a high time resolution in soft X-rays. To limit the data rates, it is essential to pre-process each frame and filter out only those pixel values that correspond to an actual photon event. The *interface controller* is designed to command that event pre-processing electronics (EPP) and to transfer the reduced science data via SpaceWire to the telemetry system of Simbol-X. The mission is described in chapter 2, the *interface controller* in chapter 3.

SpaceWire is an emerging standard analysed in this work. It is already used in recent space missions by ESA and NASA. In order to facilitate the development efforts for Simbol-X, a new SpaceWire-to-USB converter has been created that provides a general method for exchanging large amounts of data between hardware and software using the well-established Universal Serial Bus. The *SpaceWire* protocol is explained in chapter 4, the converter in chapter 5, and a concise introduction to the USB is given in appendix A.

Although it has been decided that there will be no phase B for Simbol-X, the present work continues to find its application in future missions such as the upcoming *International X-ray Observatory* (IXO).

Chapter 1

X-ray Emission in the Universe

Simbol-X is a satellite space mission for imaging and spectroscopy in the soft to hard X-ray regime of 0.5–80 keV. In this chapter some of the emission mechanisms and sources that Simbol-X could help understand are summarized.

1.1 Emission Mechanisms

1.1.1 Blackbody Radiation

A body in thermal equilibrium radiates an electromagnetic spectrum dependent on its temperature. Ignoring surface effects specific to the material, all stationary bodies radiate the same spectrum given by Planck's Law[1]

$$B_\nu = \frac{2h\nu^3}{c^2}\frac{1}{e^{h\nu/kT}-1},$$

or rewritten in terms of the photon energy $E = h\nu$

$$B_E = \frac{2E^3}{h^3c^2}\frac{1}{e^{E/kT}-1}.$$

Dependent on the temperature, the maximum of the spectrum is at the photon energy

$$\begin{aligned} E_{max} &= 2.821\,kT \\ &= 2.431 \cdot 10^{-7}\,\mathrm{keV}\left(\frac{T}{\mathrm{K}}\right), \end{aligned}$$

with the Boltzmann constant $k = 1.3806 \cdot 10^{-23}\,\mathrm{J/K}$. Temperatures around $10^5\,\mathrm{K}$ peak around $0.024\,\mathrm{keV}$, contributing with an exponential tail to soft X-rays.

1.1.2 Inverse Compton Scattering

In 1923 A. H. Compton observed the scattering of a photon off an electron, now called the *Compton Effect*. Removing the final 4-momentum of the electron, 4-momentum conservation yields the relativistic equation

$$p^\mu \cdot p'_\mu = q^\nu \cdot \left(p_\nu - p'_\nu\right),$$

where p and q are the 4-momentum of the photon and electron before the collision, p' that of the photon after the collision. This equation is valid when $\gamma\hbar\omega \ll m_e c^2$, that is, the energy of the photon as seen from the electron is much less than the rest energy of the electron.

The process is called Compton scattering, when the photon looses energy to the electron, e.g. when the electron is initially at rest and then moving. *Inverse* Compton scattering is when the electron population is at a much higher energy than the photons. In that case, low energy photons are up-scattered to higher energies

$$E' \simeq \gamma^2 E,$$

where γ is the time component of the 4-velocity better known as the gamma-factor of the electron, and E and E' are the energies of the photon before and after the collision. For a highly relativistic population of electrons this can up-scatter photons to keV and even TeV energies.

1.1.3 Synchrotron Radiation

Synchrotron radiation is produced when charged particles such as electrons are accelerated in a magnetic field, following the field lines in a helix-like pattern due to the Lorentz force

$$\frac{dp^\mu}{ds} = qF^{\mu\nu}v_\nu,$$

where ds is the spacetime interval along the path, q is the charge, F is the electromagnetic field, and v the 4-velocity. In radio a highly polarized beam can be observed. At high energies the spectrum is a power law[13]

$$B \propto E^{-\alpha},$$

where α is in the range $0.5 - 1$.

1.1.4 Bremsstrahlung

Charged particles emit electromagnetic radiation when accelerated. The power emitted by a single charge is given by Larmor's equation

$$\frac{dE}{dt} = \frac{2\,q^2\dot{v}^2}{3\,c^3}.$$

This occurs for instance in a hot plasma where electrons are accelerated in the field of the ions. The resulting spectrum features an exponential cut-off towards higher energies.

1.1.5 Emission and Absorption Lines

Discrete emission and absorption lines can be observed in the spectrum for atomic transitions. For X-ray astronomy the K_α and K_β fluorescent lines of iron with energies of 5.9 keV and 6.49 keV are of particular interest, where an electron transitions into the K shell from the L or M shells.

The nuclear decay of ^{44}Ti into ^{44}Ca also produces discrete emission lines particularly at 68 keV and 78 keV, which is important for understanding the origin of calcium in supernova remnants.

1.2 Astronomical Sources of X-rays

This section provides an overview over some of the more common X-ray emitting objects.

1.2.1 Active Galactic Nuclei

Many if not all galaxies feature a central black hole with masses of 10^6–$10^9\,M_\odot$. Whereas the black hole in the center of the milky way is relatively quiet, *Active Galactic Nuclei* (AGN) accrete 1–3 $M_\odot$ every year, producing accretion discs and massive outflows in the form of jets that can span thousands of parsecs.

AGN frequently radiate at or above the Eddington luminosity given by the maximum that a spherically symmetric accreting object of mass M can radiate:

$$L_{\mathrm{Edd}} = \frac{4\pi c\,G\,M\,m_p}{\sigma_T},$$

where m_p is the mass of a proton and σ_T is the Thomson cross section for the collision between a photon and an electron. It is given when the

Figure 1.1: This is a composite image of Centaurus A at a distance of 3.4 million parsec. In blue is the X-ray image from Chandra, in orange the submillimeter taken with the Atacama Pathfinder Experiment, and in white and brown the optical view from the Max-Planck telescope. (Credit: X-ray: NASA/CXC/CfA/R.Kraft et al.; Submillimeter: MPIfR/ESO/APEX/A.Weiss et al.; Optical: ESO/WFI)[6].

gravitational force and the radiation pressure on the infalling material are equal. In AGN, the material is not spherically falling onto the central black hole, but due to angular momentum and friction it is forming a hot accretion disc with temperatures exceeding 10^5 K.

AGN are classified into three basic types Seyfert 1, Seyfert 2, and Blazars. These are distinguished by the visibility of the nucleus and our viewing angle. The nucleus of Seyfert 1 galaxies is mostly absorbed by a gaseous ring. In Seyfert 2 galaxies the view to the nucleus is unobstructed, whereas with blazars we are looking directly into the jet. AGN show a time variability of their luminosity on the order of hours, resulting in an estimated size of a few astronomical units.

Shown in Figure 1.1 is an X-ray–submillimeter–optical composite of the Seyfert 1 type galaxy Centaurus A. Much of the optical range is absorbed by a ring of gas. The submillimeter radio observation is mostly associated with the outer lobes at the end of the jets, and the X-ray observations show the jet where it is still highly relativistic and the energetic inner regions of where it is produced.

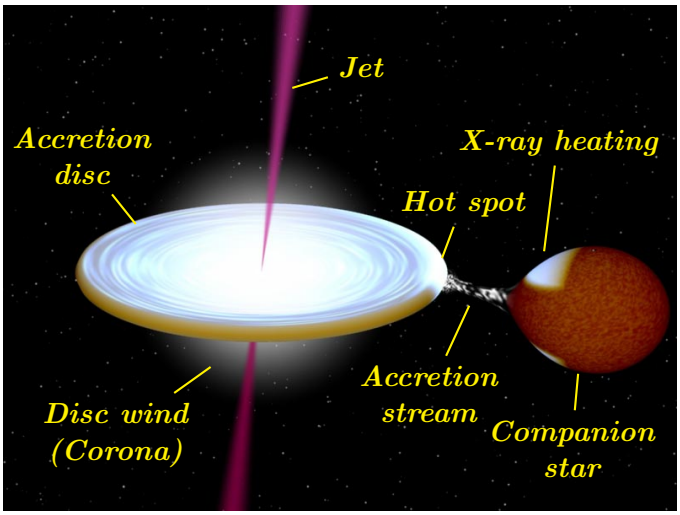


Figure 1.2: An artists impression of the microquasar GRS1915+105. Matter is accreted by the compact black hole via Roche-Lobe overflow from the companion star forming a hot spot where the matter hits the accretion disc. [3]

1.2.2 Cosmic X-ray Background

The *cosmic X-ray background* (CXB) peaks around 30 keV. Its origin is still unclear. One possible source could be a population of AGN with a hard spectrum that are too faint to detect in soft X-rays with Chandra or XMM-Newton. Simbol-X could be able to resolve those and contribute to the understanding of the CXB.

1.2.3 X-ray Binaries

In X-ray binaries a neutron star or black hole orbits a companion with an orbital period of typically a few days. When the companion star is so large that its surface is close to the Lagrange point L_1, matter may flow from the star to the compact object via Roche-Lobe overflow, as pictorially displayed in Figure 1.2 for GRS 1915+105. Thermal X-ray emission is produced by the accretion disc close to the black hole reaching temperatures up to 10^5 K. The jet is primarily associated with radio waves, but closer to the black hole the electrons are relativistic enough to produce synchrotron radiation in the X-ray band, while farther out the emission reduces to infrared and radio. X-ray emission is also produced via inverse Compton up-scattering of low energy photons in the hot corona and directly in the jet. Should the compact object be a neutron star, X-rays may also be produced when the

matter falls in an accretion column onto the surface of the star.

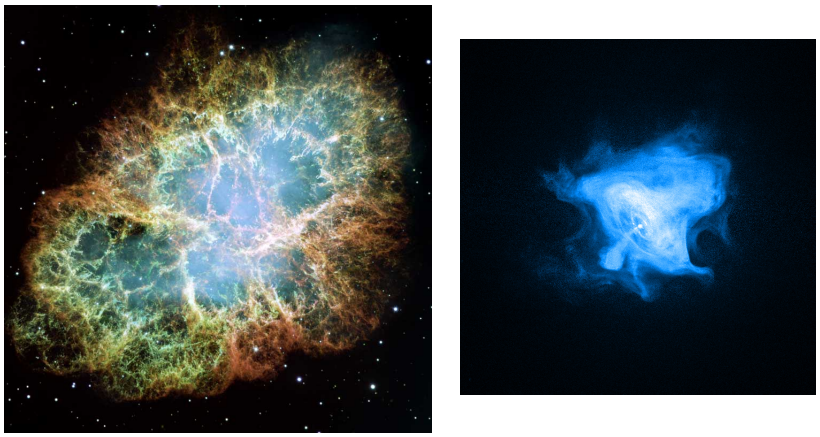
Emission lines from highly ionized iron have been observed. Their energy profile suggests that they are originating close to the compact object, since they are Doppler shifted, beamed, and anti-beamed by the rotation in the accretion disc, and red shifted by the gravitational field. However, the resolution of the spectra taken with current missions is not very satisfying. With the higher sensitivity of Simbol-X, much better spectra could be obtained.

1.2.4 Supernova Remnants

Supernova remnants are another source of X-rays where the Simbol-X mission could contribute to their understanding. In the constellation of Taurus a supernova exploded that was observed in 1054 A.D. and left behind the Crab nebula with a fast rotating neutron star, the pulsar, in the center, see Figure 1.3. The material outflow is directed by strong magnetic fields towards the poles and the equatorial plane of the pulsar. The X-ray radiation is produced by a shock where the ejected material from the pulsar interacts with the surrounding gas, and by the beamed synchrotron radiation of electrons following the magnetic field lines.

1.2.5 Intracluster Gas

A galaxy cluster typically consists of only about 1% matter visible in the optical range. Another 5–15% is hot intracluster gas with temperatures of 10^7–10^8 K that can be seen in X-rays, see Figure 1.4. The rest of the mass is called *dark matter* the nature of which is still unknown.



(a) HST mosaic (b) Chandra image

Figure 1.3: The Crab nebula (a) in an optical mosaic by the Hubble Space Telescope and (b) in soft X-rays by Chandra. The images are approximately at the same scale and orientation. (Images courtesy of NASA/STScI[11] and NASA/CXC/SAO[10]).

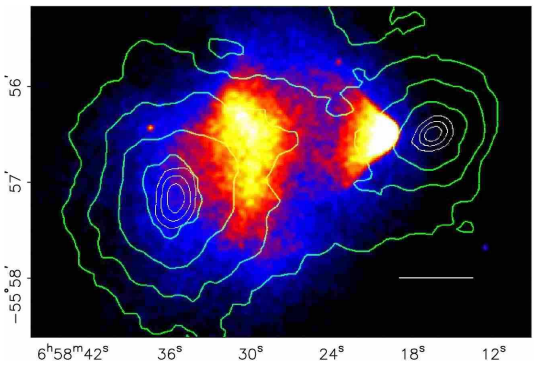


Figure 1.4: The image shows the merging galaxy cluster 1E0657-558 as observed by Chandra. The more massive eastern cluster has a temperature of 14 keV *while the less massive about* 6 keV*. The contours show the mass distribution of the clusters as determined by weak gravitational lensing.[2]*

Chapter 2

The Simbol-X Mission

In this chapter a brief introduction to the Simbol-X mission in context of other astronomical instruments of the past is given.

2.1 The Rise of Astronomical Instruments

With the invention of the telescope by Lipperhey at the beginning of the 17^{th} century, the examination of the celestial sphere could be carried out in unprecedented detail giving rise to the discovery of the moons of Jupiter when Galileo Galilei first pointed his telescope at the sky. Subsequent refinements in telescope technology further contributed to the establishment of the Copernican world view that overthrew that of the late ancient Greeks. With the invention of the Newtonian reflector, much larger apertures became possible like Herschel's telescope that was finished in 1789, and the 5 m Palomar Observatory finished in 1948.

It is only in the 20^{th} century that astronomical observations in other wavelengths could be performed. It started in 1931 when Karl Jansky of Bell Labs discovered the center of the milky way in radio waves.[15]

However, since much of the electromagnetic spectrum is blocked by the atmosphere, longer observations in the infrared, ultraviolet, X-ray, and gamma-ray bands only became possible with the advent of space flight, although modern ground based Cherenkov telescopes use the earths atmosphere as a detector medium for ultra high energy gamma-rays.

2.2 Previous X-Ray Missions

First X-ray observatories were carried with balloons and rockets in the 1960s. The following missions mark the development of space-borne X-ray observatories from its beginning in the second third of the 20^{th} century up to the present.

- The first satellite dedicated to X-ray astronomy is *UHURU* launched in 1970 from Kenya. It was sensitive to about 0.5 mCrab in the 2–20 keV range at an angular resolution of about $30'$.
- *Einstein* was the first fully imaging X-ray satellite launched in 1978. It had an angular resolution down to $2''$ in the soft X-ray range 0.15–3 keV.
- The *Roentgen Satellite* (ROSAT) from 1990–1999 conducted an all-sky survey in the soft X-ray regime.
- The *Rossi X-ray Timing Explorer* (RXTE) launched in 1995 provided astronomers with a high time resolution of up to 1 µs at 2–250 keV.
- With the launch of *BeppoSAX* in 1996, a satellite with an angular resolution of $9.7'$, imaging capabilities were available up to 10 keV.

Only RXTE and the following more recent missions are still in operation today:

- The *Chandra X-ray Observatory* is characterized by a high angular resolution of almost $0.5''$ in the 0.1–10 keV range. It was launched in 1999.
- Launched in the same year as Chandra, the *X-ray Multi-Mirror Mission* (XMM-Newton) has a lower angular resolution of $6''$, but a much higher sensitivity. Its energy range is slightly larger from 0.1–15 keV.
- Onboard the *International Gamma-Ray Astrophysics Laboratory* (INTEGRAL) is the Joint European X-ray Monitor sensitive up to 35 keV at an angular resolution of $3'$. Also onboard is an instrument with $12'$ resolution starting at 15 keV. It was launched in 2002.
- Finally, *Suzaku* contains a spectrometer for soft X-rays with a spatial resolution of $1.8'$, and a non-imaging detector for hard X-rays.

With Chandra and XMM-Newton two similar missions with a high angular resolution in soft X-rays below 10 keV are available that have contributed greatly to our understanding of soft X-ray sources up to about 15 keV. These

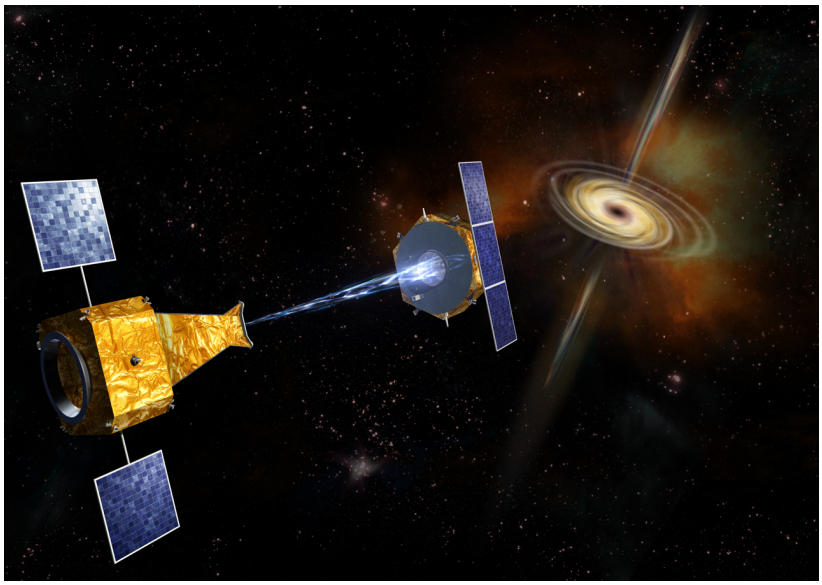


Figure 2.1: An artists expression of the Simbol-X mission. (Credit: CNES / Oliver Sattler)

missions provide accurate data in the soft X-ray regime. Missing from this list is a detector system with a high angular resolution better than 20″ and a high sensitivity in the hard X-ray band. Simbol-X intends to fill that gap.

2.3 Simbol-X

Simbol-X is an astronomical space mission in the low to hard X-ray regime. Started as a French-Italian cooperation, Simbol-X is a two-spacecraft mission, where one satellite carries a focusing Wolter optics and the other the X-ray detector positioned in the focal plane. A graphic is shown in Figure 2.1. Its launch was planned for 2014.

It is unique in that it would be the first instrument with a good spatial resolution and sensitivity in the hard X-ray range above 10 keV.

In the following, a very short introduction to the satellite's optics, detector system and electronics is given.

2.3.1 Wolter Optics

X-rays cannot be focused with ordinary lenses or mirrors. However, total internal reflection occurs at angles of the order of 1°, less for harder radiation. A Wolter optics is a mirror arrangement named after its inventor Hans

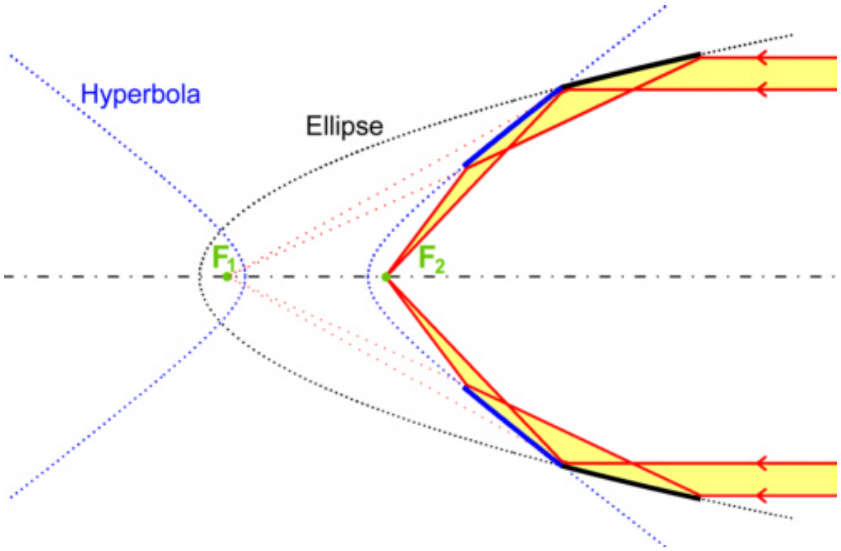


Figure 2.2: Wolter type I optics. (Image from [12])

Wolter (1911–1978) that can be used to focus X-rays using total internal reflection.

In principal, a Wolter type I optics consists of 2 mirrors, an ellipse and a hyperbolic as shown in Figure 2.2. A type I Wolter optics can be stacked many times as concentric shells that together yield a larger total aperture. Due to the nature of total internal reflection of X-rays, a Wolter optics system has the following properties that determine its geometry.

- The small reflection angle results in a large focal length, especially for hard X-rays. To achieve that, the Simbol-X optics are on a separate spacecraft flying in formation with the detector unit at a distance of 20 m as shown in Figure 2.1.
- The variance of the maximum total internal reflection angle with the photon energy results in a smaller effective aperture for harder X-rays as the outer shells no longer focus.

To achieve the scientific mission, Simbol-X was designed with 100 concentric shells, the largest 65 cm and the smallest 26 cm in diameter, with a focal length of 20 m.

2.3.2 X-ray Detector System

The detector system in the focal plane consists of two 128×128 X-ray detectors. The low energy detector (LED) becomes transparent for higher

energy photons, so it can be positioned in front of the high energy detector (HED) as shown in Figure 2.3. The two detectors are surrounded by an anticoincidence detector (ACD) for filtering particle background events and thus significantly reducing the background. The Simbol-X low energy detector is in so far unique in that it would be the first with an active anticoincidence shield. In addition, it was discussed to use a proton deflector to reduce the background from low energy protons funneled onto the detector by the optics. This would also reduce the degradation of the detector by such protons.

The detectors have the following characteristics.

- The LED is sensitive in the range 0.5 keV–20 keV. A pixel of the matrix consists of concentric drift rings collecting the electrons at the center where a DEPFET (*Depleted P-channel Field Effect Transistor*) is used to detect the amount of electrons freed by the incoming photon. Each pixel has a size of 625 µm × 625 µm. It is read out at a rate of 8000 frames per second.
- The HED has a range up to 80 keV in hard X-rays. The detector material is made up of CdTe.
- Together, the two detectors give a nearly 100% quantum efficiency across the X-ray range up to 80 keV as shown in Figure 2.4.
- The ACD catches particles from everywhere except where the photons from the optics are coming in. Comparing the time of an event in the ACD with the time of pixels from the LED and HED allows to classify these as background events and be discarded for most science measurements.

The LED, HED, and ACD are connected via a SpaceWire link to the central DPDPA processing board as shown in Figure 2.5. The software running on the DPDPA CPU board has access to a several hundred megabyte large mass memory. It will compute offset and threshold maps, command each detector to start and stop sending science data, and correlate events from the anticoincidence detector with the pixel events from the detectors. From the DPDPA board science and housekeeping data are transferred to the spacecraft telemetry system and from there to ground.

The LED electronics (LEDE) are shown in Figure 2.6. It is the purpose of the interface controller developed in this work to provide the LED with a SpaceWire interface to the DPDPA board, and to convert the science data into a standardized format.

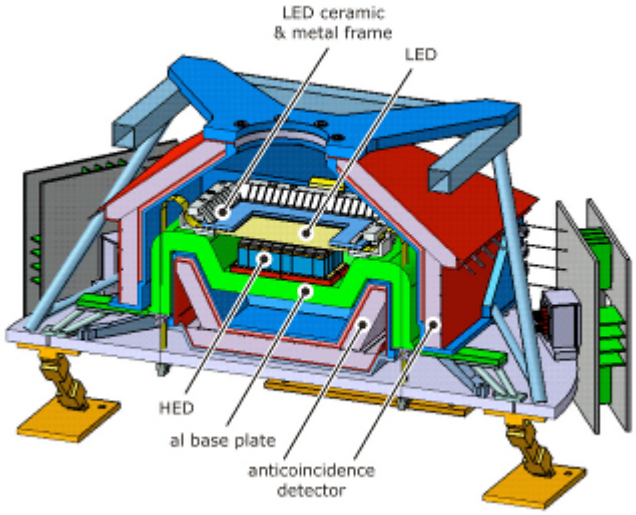


Figure 2.3: The Simbol-X detector geometry. X-rays coming in from the top hit the low energy detector (LED) first. Higher energetic X-rays continue through to the high energy detector (HED). For detecting particles from the sides or bottom, the entire detector system is surrounded by an anticoincidence detector. An aluminium base plate transports heat to the outside to keep the detector temperature below $-40°\mathrm{C}$. *(Credit: CEA / Jerome Martignac)*

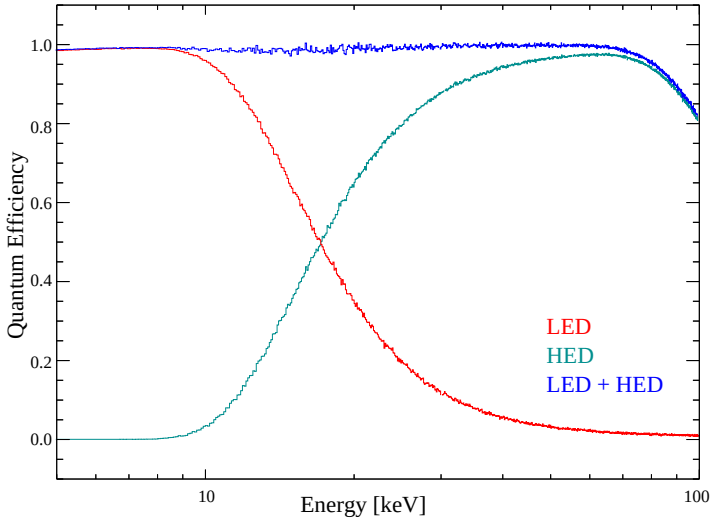


Figure 2.4: The combined quantum efficiency of the LED and HED add up to almost 100% *as shown in this simulation by C. Tenzer.*

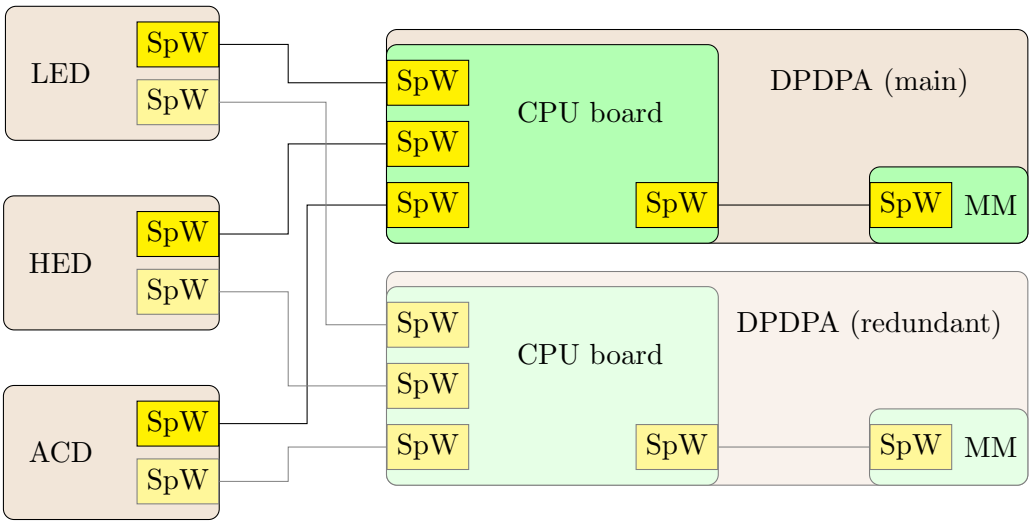


Figure 2.5: Simbol-X detector connections. The CPU board is connected via SpaceWire links to each of the three detectors, and has access to a mass memory unit (MM), which is also part of the DPDPA. Every detector has a redundant SpaceWire interface controller that connects to the redundant DPDPA shown subdued in the figure. Not shown is the connection with the telemetry electronics. (adapted from Simbol-X Detector Payload SpaceWire Utilisation Requirements*[4] Figure 1-1)*

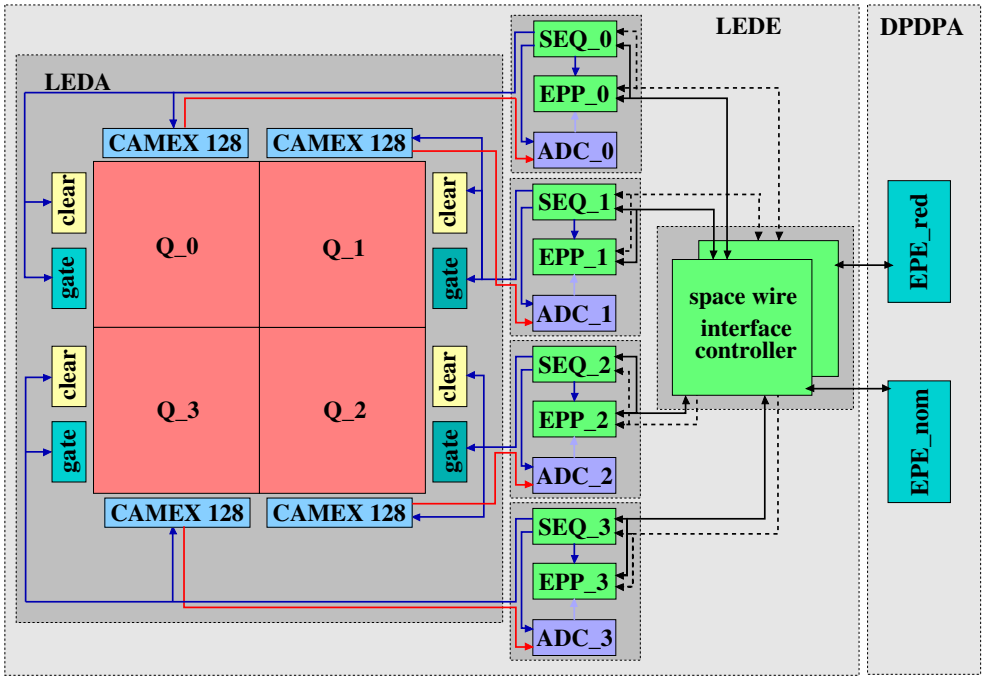


Figure 2.6: LED electronics overview. The 128×128 *detector matrix is split in 4 quadrants* Q_0*,* Q_1*,* Q_2*, and* Q_3*. Each quadrant is controlled by two switchers, gate and clear. The gate switcher enables an entire row of a quadrant, so that the energy deposited in each pixel can be read out by the CAMEX (*Charge Amplifying Multiplexer*). The clear switch flushes the electrons in a row of DEPFETs for the next frame. The analog pixel values from the CAMEX are transferred to an ADC for each quadrant, where the event preprocessor (EPP) filters for the pixels corresponding to a photon event. The switchers, CAMEX, ADC, and EPP receive their timings from the sequencer (SEQ) of each quadrant. The interface controller transfers the data from each quadrant to the DPDPA, and accepts commands to power up and configure the low energy detector electronics. Should the nominal interface controller fail, the DPDPA can enable a redundant one. (Graphic courtesy of Thomas Schanz)*

Chapter 3

The Interface Controller

The purpose of the interface controller is to provide a simple and standardized interface to the low energy detector electronics of Simbol-X. The design is realized in an FPGA with input and output ports for each of the components of the detector readout electronics. In this chapter the requirements on the interface controller are described, then the overall design principles, and finally the specifics of each component.

3.1 Interface Controller Requirements

The interface controller (IFC) provides the low energy detector of Simbol-X with a SpaceWire interface to the rest of the spacecraft, the DPDPA. As such, it needs input and output ports for every component of the low energy detector, plus those needed for the SpaceWire communication with the rest of the spacecraft.

Figure 3.1 gives an overview of the components the interface controller needs to communicate with. The following input/output port considerations need to be made for every quadrant.

- The sequencer and the EPP are combined onto the same board, so that a single SPI link is sufficient for control commands towards the detector. The SPI link is used to configure and command the sequencer and the EPP. Most importantly it is this link via which the IFC must upload offset and threshold maps for the EPP, and it needs to command the sequencer when it should start and stop operating the detector readout electronics. The SPI link needs 3 output and 1 input signal.

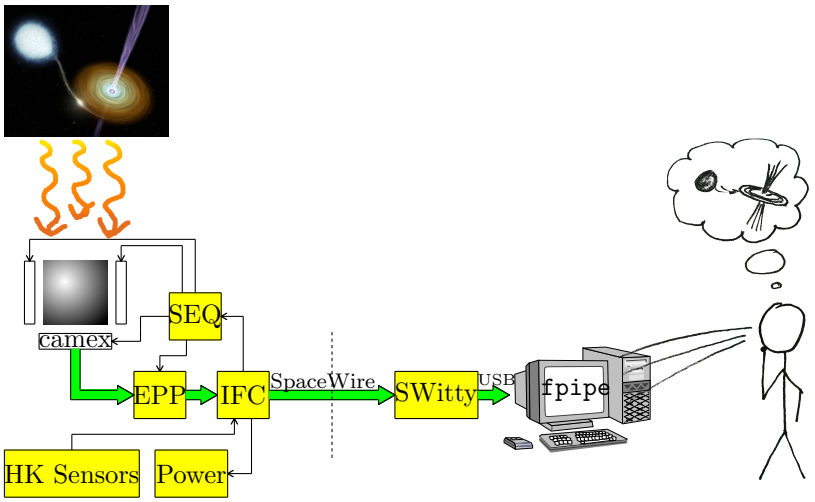


Figure 3.1: LED electronics overview. Starting at the top left, X-rays from a binary system hit the detector matrix which is above the CAMEX that amplifies the raw pixel values. To each side of the detector matrix are switchers to control the detector read out. The amplified pixel values from the CAMEX are digitalized with an ADC (not shown), and sent to the event-preprocessor (EPP). The switchers, CAMEX, and EPP are controlled by the sequencer (SEQ). The interface controller (IFC) collects the science data from the EPP, sends commands to the sequencer, controls the power, and collects housekeeping (HK) sensors data. The communication with the rest of the spacecraft is done via a SpaceWire link. Shown here is a setup as is planned in the laboratory for testing, where the SpaceWire communication is transferred by SWitty, explained in chapter 5, to a computer via the USB, and analysed with the `fpipe` *analysis software.*

- The EPP reduces the detector output to only those pixels that correspond to a photon event. Each pixel is encoded in a 64-bit format that includes a time code, the position, the energy, and the type of the pixel. In order not to use up too many I/O ports of the interface controllers' FPGA, a 16-bit parallel bus is envisioned for the data transfer of the pixels from the EPP, controlled by 2 extra signals.

- The housekeeping and power sensors were not set as of this writing, but to save FPGA I/O ports, an analog multiplexer is added to the interface controller board. It directs analog sensor signals onto a single ADC that, too, will be on the IFC board.

- The interface controller must also set the power for the detector. However, the power supply interface was too early in its development to be considered for the IFC at the time of this writing.

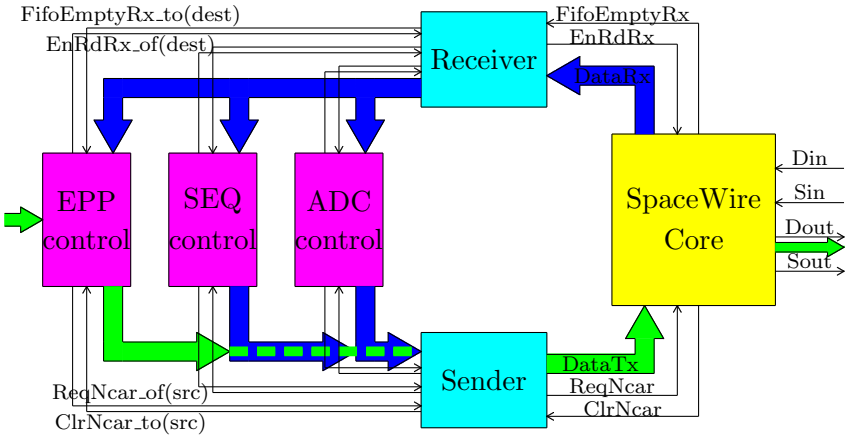


Figure 3.2: IFC internal overview. On the right, the SpaceWire core with the SpaceWire signals is shown. The `Sender` *and* `Receiver` *components distribute and collect packets to and from the* `*_control` *components that do the actual work of communicating with the components of the low energy detector. Not shown are the external signals for that communication. The thick arrows in light color depict the path of science data from the EPP through the interface controller.*

- Finally, the interface controller will be used to carry a clock signal to the EPP. However, this clock signal will not affect the IFC directly. Instead, it is distributed through signals electrically separate from the IFC.

3.2 General IFC Design Considerations

At its core, the interface controller must act as a router to the various components of the low energy detector. The basic design is shown in Figure 3.2. Control components provide the low-level interaction with each LED component. The receiver component is responsible for distributing packets from the SpaceWire core to the individual control components. Similarly, the sender is responsible for collecting the packets from the control components and sending them to the SpaceWire core.

The interface controller has undergone a complete rewrite since a first working version was in place. Compared to that, the current design has the following advantages.

- Debugging and testing of IFC components is extraordinarily easy.

- There are no complicated interrelationships between components and internal signals.
- The IFC is easily extensible.

These three advantages have resulted in a nearly bug-free implementation. They are the result of some general design principles, some of them created as a response to the first version of the interface controller, some of them well known in the software programming community. Even though that first version of the interface controller exhibited a highly modular design, it is only with the help of these guiding principles that a truly workable design could be created. Those principles shall be described in the next few sections.

3.2.1 Equality

All components are created equal. What this means is that the components are in no particular hierarchical order in and of themselves. It is only their function that determines their place inside the interface controller.

The principle of equality entails that the communication between the individual components must be standardized. Ignoring external signals, the entity of each component has been chosen to match precisely the host-side interface of the SpaceWire core. It consists of a 9-bit data bus and two signals, `FifoEmptyRx` and `EnRdRx`, for receiving packets, and a 9-bit data bus and the signals `ReqNcar` and `ClrNcar` for sending packets.

The practical implication is that each component can be connected directly to the SpaceWire core and be tested separately from all other components.

3.2.2 Uniqueness

The principle of uniqueness says that code should not be duplicated. The principle can be further interpreted to mean that the *idea* expressed within a piece of code should occur only once. Duplication early in the development means that changes in one copy might not be applied to another copy, and the relationship between the two copies becomes unclear as the code evolves. This is hampering development especially when ideas are changing, since every piece of code needs to be reviewed for subtle changes.

Uniqueness of code has played a central role from the very beginning of the development of the interface controller. Early versions, however, broke the principle with regards to the uniqueness of ideas expressed in the code. It is uniqueness of code *and* of ideas that leads to a clean modularization

framework. The principle demands that similar but distinct functions be merged together or split up further. Uniqueness is the driving force behind factorization of a design into independent modules with clean interfaces.[1]

For hardware, the situation is much reversed. Using the same components in many parts of the design means that optimization can be directed towards fewer distinct elements. Should the element be designed in a hardware description language, this leads us back to the principle of uniqueness, where concentrating on a single implementation of a component instantiated many times greatly eases development.

3.2.3 Simplicity

Simplicity is a well known term in software development. It can mean many things including minimalism, symmetry, and elegance. It also ties in very closely with the ideas of uniqueness and modularity, as both these demand that complications are coded only once, leading to a simpler overall design.

When uniqueness is impossible, such as using the same interface between all the components, then simplicity demands that this interface be a simple interface. In the case of the interface controller, it entailed forbidding complicated interrelationships between the various components, and symmetrizing the relationship between receiving and transmitting data busses.

Simplicity cannot be pinpointed easily. It pervades the design on every level. When complications are hidden away in submodules, simplicity demands an interface for these submodules that can be easily described and shared among developers. What this usually means is that each module should do only one thing, and do that well.[2]

3.2.4 Beauty

The combination of the principal ideas of equality, uniqueness, and simplicity has resulted in a highly modular design. The three principles are in many ways building on top of the usual mantra of a modular design with independent components, but in some ways they are more general. Indeed, it could well be argued that these might set the very foundation of which modularity is perhaps the best solution.

Perhaps the most important consequence is that the code in its entirety may be called beautiful. Beautiful code is simple code. Beauty is achieved when simple code snippets are combined to create a design capable of much

[1] Hence, this subsection may also be called “Modularity is not enough”.
[2] This is one of the basic philosophies behind UNIX.

Table 3.1: Transmitter and receiver host data interface coding. The MSB, e.g. `DataRx(8)`*, is leftmost, the LSB rightmost. (Table 7, page 54 in ECSS)*

`DataRx/DataTx`	Meaning
`0xxxxxxxx`	8-bit data
`1xxxxxxx0`	EOP
`1xxxxxxx1`	EEP

more than its individual components. That is, beauty ensues when removed complexity results in greater applicability. It is the beauty of small but generally applicable code that drives development. The practical result is that it is a delight to maintain and extend the design.[3]

3.3 IFC components

The interface controller consists of the SpaceWire core component, the Receiver and the Sender, and multiple control components. In this section these components of the interface controller will be described in more detail.

3.3.1 SpaceWire core

The SpaceWire core component used was developed by Frédéric Pinsard and Christophe Cara at CEA. The host-side interface exported by this component is used throughout the design of the interface controller.

There are, actually, two interfaces. One for the receiving line, another for the transmitting line, denoted by appending `Rx` and `Tx` to the respective signal names. The data is published via the signals `DataRx` and `DataTx`. These are 9-bit data busses, and as shown in table 3.1, the 9^{th} bit, e.g. `DataRx(8)`, is used to distinguish between a data character and a control character. If that bit is high, then it is a control character, and the LSB distinguishes between a normal end-of-packet and an error-end-of-packet. If the MSB is low, the remaining 8 bits constitute a data byte.

The interface controller is notified of a new character by the SpaceWire components two control signals `FifoEmptyRx` and `EnRdRx`. They are used just like the interface to a FIFO that is filled with SpaceWire characters coming over the SpaceWire link, and emptied by the interface controller. On the transmitting side, there are the two signals `ReqNcar` and `ClrNcar` with the same idea but slightly different semantics as the signals on the

[3] It is somewhat unfortunate that beautiful code is code that does not need much maintenance.

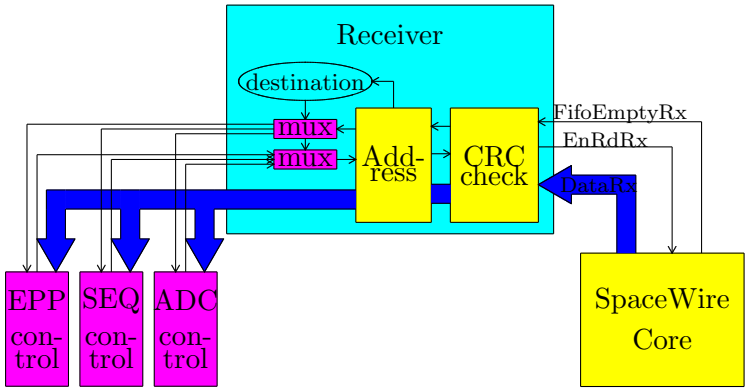


Figure 3.3: The Receiver distributes packets to their respective control component.

receiving side. To merge the two, the signal `FifoEmptyRx` should have been negated, i.e. `FifoNotEmptyRx`.

The SpaceWire component is the first to receive commands over the SpaceWire link. As such, it is viewed as the master to the other components of the interface controller.

3.3.2 Receiver

The purpose of the receiver component is to distribute incoming packets to their respective control component. In doing that, the receiver is also stripping the header and footer from each packet.

Although the packet formats were not finalized as of this writing, the header will have the same structure for all packets. It will contain a byte identifying the LED as its destination. For compatibility with specifications, another byte will specify that the packet is in a custom format. A third byte is used for the packet type distinguishing packets for housekeeping and science packets. It might also be used as a logical address for identifying the control component responsible for the packet.

The footer consists of a single CRC8 byte for checking the integrity of the packet. It is the responsibility of the receiver to check that the CRC is correct and to provide the control components only with the cargo of the packets. It does this by controlling a multiplexer via the register `destination` that transfers the `FifoEmptyRx` and `EnRdRx` signals to the specified control component. This is shown in Figure 3.3. In this way, the control components can essentially "see through" the receiver as if directly connected to the

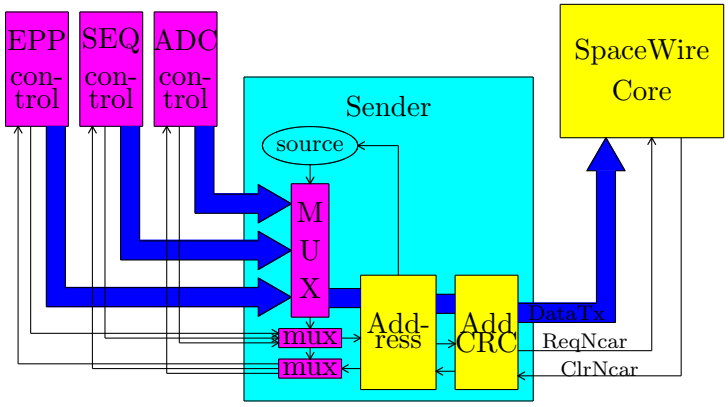


Figure 3.4: The Sender essentially multiplexes the access to the SpaceWire core.

SpaceWire core.

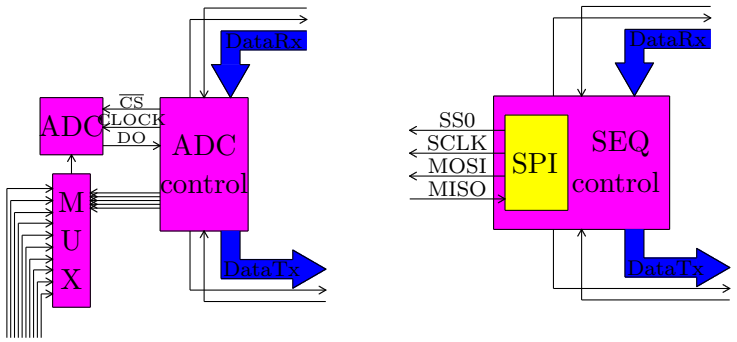
3.3.3 Sender

The sender component does the opposite of the receiver. It polls which control component has a packet to send and grants it access to the SpaceWire core, adding an appropriate header and a CRC8 footer to each packet. The internal setup of the sender is very similar to that of the receiver, and it is shown in Figure 3.4.

3.3.4 ADC_control

`ADC_control` is the prototype for the housekeeping control component. Housekeeping data includes values from temperature, voltage, and current sensors, the most prominent probably being the one reporting the temperature of the detector matrix. Which values exactly will be needed was not clear yet as of this writing and will be determined during detector operation in the laboratory.

To save ADCs and FPGA I/O ports, most analog sensor values will be multiplexed onto a single ADC. The design is shown in Figure 3.5a. The `ADC_control` component sets an address on the multiplexer, which then puts the signal from the selected sensor on the ADC input. The control component then proceeds to read out the digitalized value from the ADC via the signals shown in the figure.



(a) `ADC_control` (b) `SEQ_control`

Figure 3.5: (a) Analog sensor signals from the bottom left are multiplexed onto an ADC. The component `ADC_control` *inside the FPGA sets an address on the multiplexer, and reads out the converted value via a standard ADC interface. (b) The sequencer control component allows the DPDPA software on the other side of the SpaceWire link to send commands directly to the sequencer, providing an SPI converter as it does so.*

3.3.5 SEQ_control

This control component interacts with the sequencer. Packets coming over the SpaceWire are sent to the sequencer via an SPI bus. The interface controller acts as a kind of tunnel to the SPI interface of the sequencer, so the DPDPA software can essentially communicate directly with the sequencer.

The basic setup is shown in Figure 3.5b. Whether a more complicated setup is necessary or even desired is yet to be determined. The current setup might have the drawback of taking a long time to upload an offset or threshold map as each pixel needs to be set individually. Every command is 6 bytes long. Adding to that 3 bytes for the header, 1 for the CRC sum, and roughly half of one for the end-of-packet, adds up to 10.5 bytes per pixel. For a total of 4096 pixels, that makes at least 43008 bytes per map. At a rate of 2 MB/s that takes 22 ms over the SpaceWire. The SPI link is slower, and takes $6 \times 8 = 48$ SPI clock cycles per command. For successful operation, the SPI clock is lowered to 3.75 MHz, making 12.8 µs per command, or about 53 ms per map. Sending the 2-byte status result received while transmitting over the SPI back over the SpaceWire takes another 13.3 ms per map. In total, uploading an offset or threshold map should take about 88.3 ms. Even though there will be some extra latencies introduced that are not included in this calculation, there does not seem to be any pressing need for a more complicated setup.

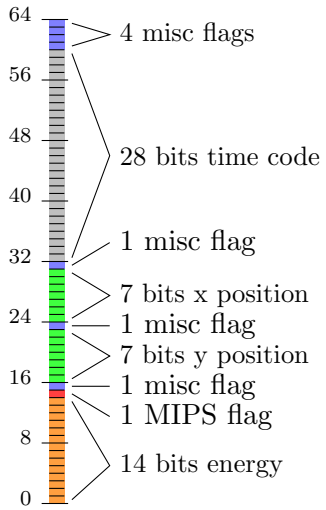


Figure 3.6: The figure shows the preliminary assignment of each of the 64 bits for every pixel. Small ticks mark bit boundaries, large ticks mark byte boundaries.

3.3.6 EPP_control

The `EPP_control` component collects the science data from the event preprocessor, packages them in a standardized format, and sends them off over the SpaceWire to the DPDPA software. The control of the EPP is done via the sequencer, and is not part of this component.

The standardized format is made up of 64 bits for every pixel. The information encoded in these 64 bits is shown in Figure 3.6 as was agreed upon at the time of this writing. The data from the EPP is very similar, and contains for each valid pixel a time code, the energy, the position, and various miscellaneous flags subject to change. The standard format has been deliberately chosen with the following points in mind.

- The time code, x, y positions, and energy are byte aligned, because that makes for much easier processing in software.
- The MIPS flag has been chosen as an extra bit for the energy, so that a simple check of the energy is all that is necessary for the DPDPA software to distinguish good events from MIPS.
- The quadrant number is within the 7^{th} bits of the x and y positions, making the position a natural number across the entire 128×128 detector matrix.

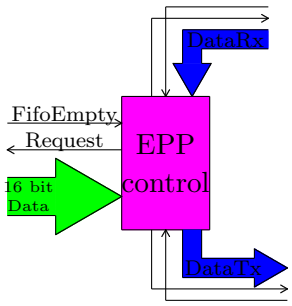


Figure 3.7: The `EPP_control` *component collects the science data from the event preprocessor over a 16 bit data bus with an interface reminiscent of a FIFO.*

- To allow for additional miscellaneous flags, only the first 28 bits of the EPP time counter are transmitted with every pixel. This would have a wrap-around time of 13.42 s. The rest of the time code can then be retrieved via other means.

The 64 bits per pixel are envisioned to be transmitted over a 16 bit bus as shown in Figure 3.7, where each signal is LVDS encoded for better electromagnetic compatibility.

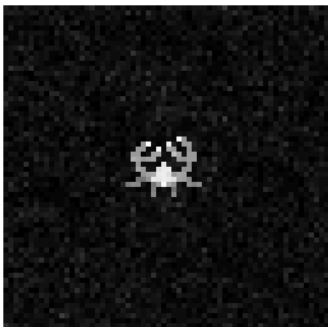
In order to test the `EPP_control` component a simulator was written. The simulator will be more closely described in section 3.4.1. Even though the data format from the simulator is somewhat different, it could be shown that the interface controller can sustain data rates well above the maximum 2 MB/s of the SpaceWire link.

3.3.7 PWR_control

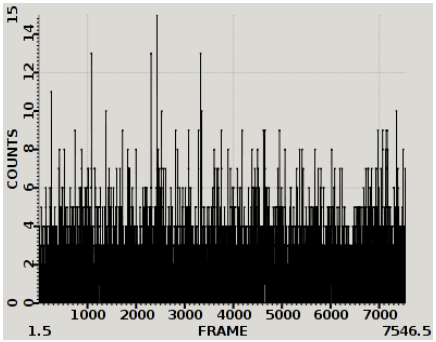
This component controls the power to the LED. It does not exist yet, as it only makes sense to develop when more details about the power interface are available. It is mentioned here as it is an important part of the interface controller should Simbol-X fly.

3.4 Additional components for testing

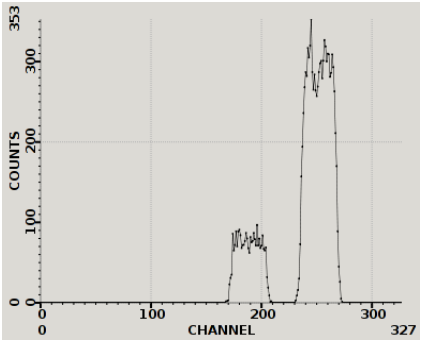
In this section two control components, `ECH_control` and `SIN_control`, and a simple EPP data simulator used for testing purposes are introduced.



(a) Intensity map



(b) Light curve



(c) Spectrum

Figure 3.8: The Crab lifts its right claws.

3.4.1 EPP data simulator

At the time of this writing, the hardware to connect the EPP with the interface controller was not available. As a consequence a simple simulator for the data was written as a subcomponent of the `EPP_control` module. The raw pixel energy values are created from three sources listed below. In the following, `data` is a VHDL variable holding the raw pixel value, and `rand` is a pseudo-random number generated from a 63 bit linear feedback shift register (LFSR). It has a 5 bit range, that is from 0 to 31.

- Offset: `data <= 0x133 + rand;`

  The offset is valid for every pixel. It corresponds to the current measured in the detector when there was no photon event.

- Image: `data <= 0x1F0 + rand;`

  To make things more interesting, a PNG file containing a picture of a well known X-ray source is loaded at VHDL synthesis time. The RGB value of each pixel in the image determines the intensity. The resulting image is shown in Figure 3.8a.

- Background: `data <= 0x22F + rand;`

  The background is also valid for all pixels, but it is not set every single frame. Instead there is a randomization resulting in about 2 to 3 pixels per frame contributing to the background.

An intensity map, a light curve, and a spectrum in arbitrary units are shown in Figure 3.8 as screen shots from the `fpipe` analysis software.

The EPP data simulator acts as a source of data. The rate of data transmission can be throttled if desired or needed. The simulator has made development somewhat of a delight, and the slightly asymmetric nature of the image proves that the entire chain from the interface controller to the analysis software is interpreting the X and Y coordinates of the pixels correctly.

3.4.2 ECH_control

The `ECH_control` component echoes everything it receives back over the SpaceWire. It is very handy for testing the SpaceWire connection in all rigor. Its main use is for testing when the SpaceWire core is replaced by SWitty, which is a SpaceWire-to-USB converter described in chapter 5.

3.4.3 SIN_control

The `SIN_control` component acts as a sink accepting all data, sending none. It is used to test the SpaceWire link or SWitty unidirectionally.

3.5 Summary

The basic function of the interface controller is to be a router that distributes packets coming in over a SpaceWire connection to the individual components of the low energy detector. This function has been nicely tucked away in two components, a receiver and a sender. For every component of the low energy detector, a control component has been developed when possible.

The interface controllers' design in Figure 3.2 has proven itself to work rather well. Simplicity and uniqueness of code and ideas throughout have contributed to its versatility. Simple debugging was achieved by creating all top-level components as equals, allowing each to be connected directly to the SpaceWire core. Beauty in logic and function in combination with equality, uniqueness, and simplicity has resulted in a highly modular design with great independence for every module.

Multiple additional control components can be easily added. As a consequence, the interface controller has proven to be simple to use yet very versatile even in unforeseen situations such as a testing device for SWitty, the SpaceWire-to-USB converter described in chapter 5. As such, it will be interesting to see how the integration with the other components of the low energy detector of Simbol-X will go.

Chapter 4

The SpaceWire Protocol

A SpaceWire link is a full-duplex serial communication line capable of speeds of 2 Mbit/s up to 400 Mbit/s. The standard was spun off the IEEE 1355-1995 standard with the goal to make it suitable for space applications. The specification was first released on January 24, 2003 by the *European Cooperation for Space Standardization*, and it is now available under the designation ECSS-E-ST-50-12C(31 July 2008) at `http://spacewire.esa.int`[7].

It is already used in missions such as the Herschel space telescope launched in 2007 and the SWIFT space observatory launched in 2004. In the present context of Simbol-X, SpaceWire cables are used to connect the individual detector electronics to a common processing board known as the DPDPA. It is the purpose of the interface controller to provide the low energy detector of Simbol-X with a SpaceWire interface. In this chapter the SpaceWire protocol is described in some detail.

4.1 SpaceWire Basics

SpaceWire cables are specially shielded against electromagnetic interference. Since interference will most likely occur equally on all wires in the cable, only the difference between two signals is used. Such *differential signaling* is also standard in most serial communication protocols including the USB, Firewire, or Ethernet, although there are major differences in the electrical details. For SpaceWire, low voltage differential signaling (LVDS) is used, where the information is carried over the wire via a current that produces a low voltage signal across a 100 Ω resistor at the receiver. These electromagnetic interference counter measures result in a very reliable connection even at 60 Mbit/s as used in the laboratory.

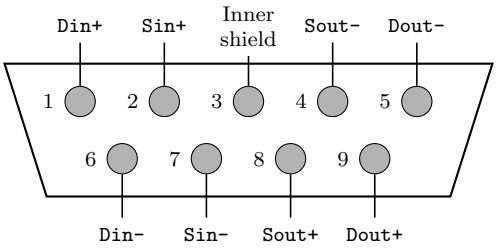


Figure 4.1: SpaceWire connector contact identification viewed from rear of receptacle or front of plug. (adapted from Fig. 10, page 40 in ECSS-E-ST-50-12C[7])

SpaceWire is a fairly simple standard with a low footprint on electronic and energy resources. However, there is little guarantee for data integrity. A CRC8 field at the end of each packet was chosen for the Simbol-X mission to be able to detect data corruption and request resending of the data. This is sufficient for single bit errors.

The SpaceWire protocol also features what is called *time-codes*. These are data characters that take priority over normal data characters, and can be sent over a SpaceWire link at any time. With a relatively simple extension to the SpaceWire protocol proposed by F. Pinsard and C. Cara[14], time synchronization via time-codes can be achieved to an accuracy of up to two clock cycles. Although time-codes are not used at all for Simbol-X, they are mentioned here for completeness, and may be used in future space missions.

SpaceWire is a packet oriented protocol. A packet can be sent over a link only one at a time. To allow for smaller packets, SpaceWire encodes end-of-packet characters efficiently as a 4-bit character, while normal characters are 10-bit with 8 bits of data. There is no start-of-packet character. The end of one packet marks the beginning of the next.

4.1.1 Basic Signals and Connectors

A SpaceWire link consists of 9 wires with micro-miniature connectors at each end. The connector identification is displayed in Figure 4.1.

There is an input and an output signal. The input signal is encoded using data-strobe, described in section 4.1.3, using a `Din` and an `Sin` signal. The `Din` and `Sin` signals are transmitted via LVDS, see section 4.1.2, giving `Din+`, `Din-`, `Sin+`, and `Sin-` signals. The same applies to the output signals, resulting in the signals `Dout+`, `Dout-`, `Sout+`, and `Sout-`.

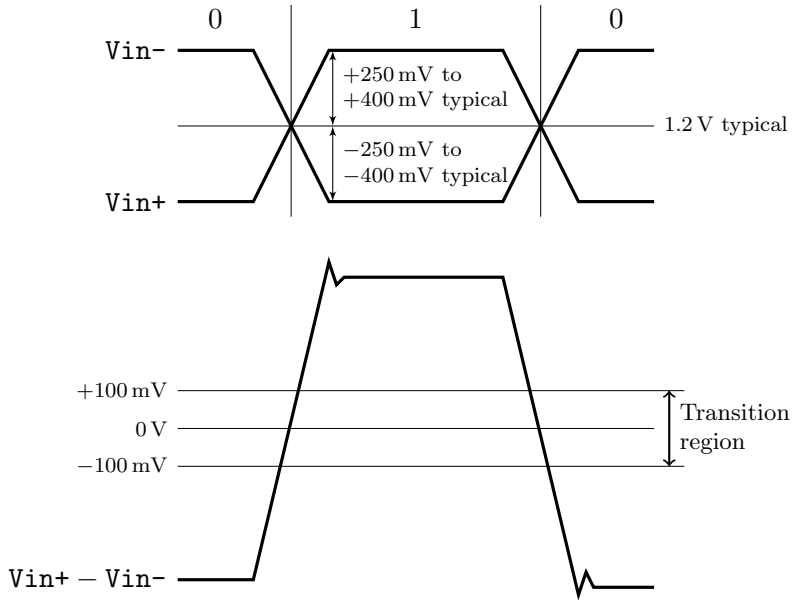


Figure 4.2: LVDS signaling levels. The above figure shows the absolute values of the voltage of the positive line `Vin+` *and the negative line* `Vin-` *of a differential signal, and their swing around the* 1.2 V *value. The lower figure shows the difference between the two. A transition is detected when the difference is within* 100 mV *of* 0 V*. (adapted from Fig 3, page 29 in ECSS-E-ST-50-12C[7])*

4.1.2 Low Voltage Differential Signaling

Low voltage differential signals are used for all signals because of their high immunity to induced noise. The voltage swing used here is typically ±350 mV, although it can go from 250 to 400 mV, see Figure 4.2.

The transmitter provides a constant 3.5 mA current when there is a 100 Ω termination resistance on the other end. The receiver must have a high input impedance, so that most of the current flows through the termination resistor.

4.1.3 Data Strobe Encoding

For each line there is a data signal D and a strobe signal S. The strobe signal changes whenever the data signal does not change, as shown in the example in Figure 4.3. In this way, only transitions need to be detected, resulting in an increased skew tolerance between the data and strobe signals. In case needed, the clock is reproduced by the XOR of the `D` and `S` signals.

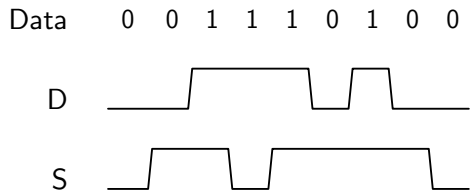
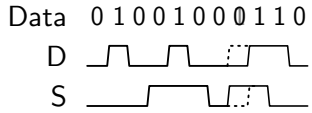


Figure 4.3: Data Strobe (DS) encoding. The strobe signal `S` *changes whenever the data signal* `D` *does not. (adapted from Fig 5, page 30 in ECSS-E-ST-50-12C[7])*

The robustness of the data-strobe encoding has been tested by instantiating two cores in a single FPGA. Glitches were then inserted on the FPGA-internal D and S lines. There are two kinds of glitches that can be tested in this way. Either there is a bit flip in one of the signals, or there is a bit flip in both of them. As shown in the following, these errors are handled very well by the data-strobe encoding.

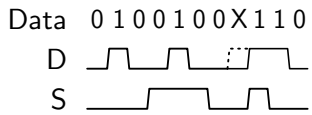
Let us first consider a bit flip on both lines, the D and the S line.



In dashed lines, the original example is shown, the solid line shows the signal with the glitch on both lines. This results in a signal that is still a valid data-strobe encoded signal, but with one bit flipped.

In general, if both the D and S lines flip, the transition will be on the other line, so the signal will remain valid, though with one bit flipped. One-bit flips are easily detected via the parity bit, described later in section 4.1.5.

The second kind of error is where only one of the lines flips. Let us consider the example where the D line flips one bit. This is the same situation as when the skew between the data and strobe signals becomes too large.



For a successful decoding it is essential to determine the order of transitions. The SpaceWire core must determine which signal transition is occurring before the other. In the extreme case discussed here, two transitions are occurring at the same time, with differences introduced by jitter inside the

FPGA. The SpaceWire receiver must tolerate this and can randomly choose which is first.

The dashed signal shows the original data signal in the figure above. Should the actual transition occur after the transition in the strobe signal, or be detected as such, then the bit marked 'X' will be a 0. This error is once again detectable via the parity bit.

Should the actual transition occur slightly before the transition in the strobe signal, then 'X' is 1, and there is no error.

This epitomizes the extraordinary skew and jitter tolerance that the data-strobe encoding brings to the SpaceWire protocol.

4.1.4 Characters and Control Codes

The SpaceWire protocol is a packet oriented protocol, and it uses a special character encoding for marking the end-of-packet or error-end-of-packet characters.

A character consists of either 4 or 10 bits. The first two bits in a character are always the parity bit and the control flag. The parity bit is used to detect single bit errors, and it is set as described in Section 4.1.5. The control flag determines whether the character is a data character or a control character.

When the control flag is 0, then the character is a data character, and the next 8 bits contain the data, LSB first.

If the control flag is 1, then the character is a control character, and the next 2 bits indicate whether it is an FCT (flow control token), EOP (end of packet), EEP (error end of packet), or ESC (escape) character.

In addition, the SpaceWire protocol recognizes combinations of characters, called *control codes*. There are two control codes, NULLs and Time-Codes. A NULL is also called a NULL-*character*. It is formed from an ESC character followed by an FCT. A time code is formed from an ESC, followed by a data character.

Figure 4.4 illustrates the individual character types and control codes. Data characters, EOP, and EEP characters are also called normal characters, or N-Chars. ESC, FCT, NULL-characters, and time-codes are also called link characters, or L-Chars.

4.1.5 The Parity Bit

The purpose of the parity bit is to detect simple 1-bit errors. It is set such that the total number of 1's in the covered field is always odd.

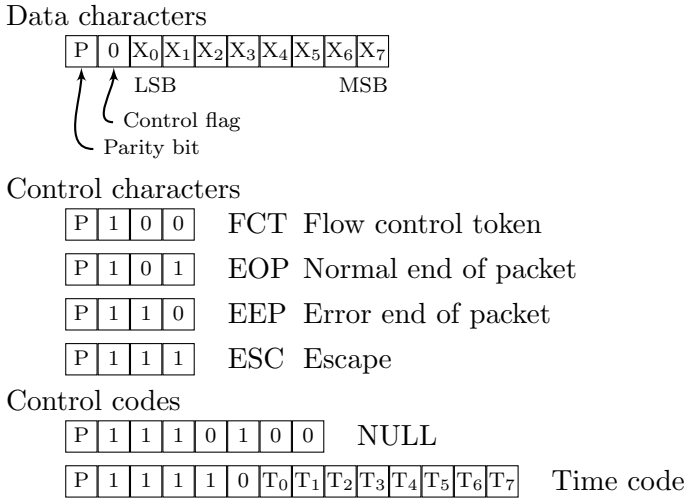


Figure 4.4: Data characters, control characters, the null character, and time codes. (adapted from Fig 6, page 31 in ECSS-E-ST-50-12C[7])

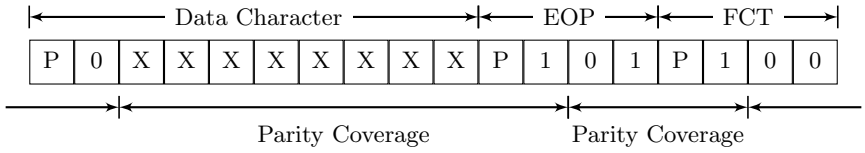


Figure 4.5: Parity coverage. As an example, the sequence of a data character followed by an EOP and an FCT is shown. The parity bits are set such that the number of 1's in the parity coverage is odd. The figure illustrates that the parity coverage includes the end of the previous character and the beginning of the current character. (adapted from Fig 17, page 53 in ECSS-E-ST-50-12C[7])

The parity bit covers the following bits: itself, the control flag, and the last 8 bits of the previous data character or the last 2 bits if the previous character was a control character. Figure 4.5 illustrates the coverage of the parity bit.

4.1.6 Packets

A packet is formed by one or more data characters followed by either EOP or EEP. An EEP indicates the end of a packet in which an error occurred.

Characters from two different packets may not be interleaved. Only FCTs, NULLs, and Time-Codes can be interleaved with a packet. The data character immediately following an EOP or EEP is the first character of the next

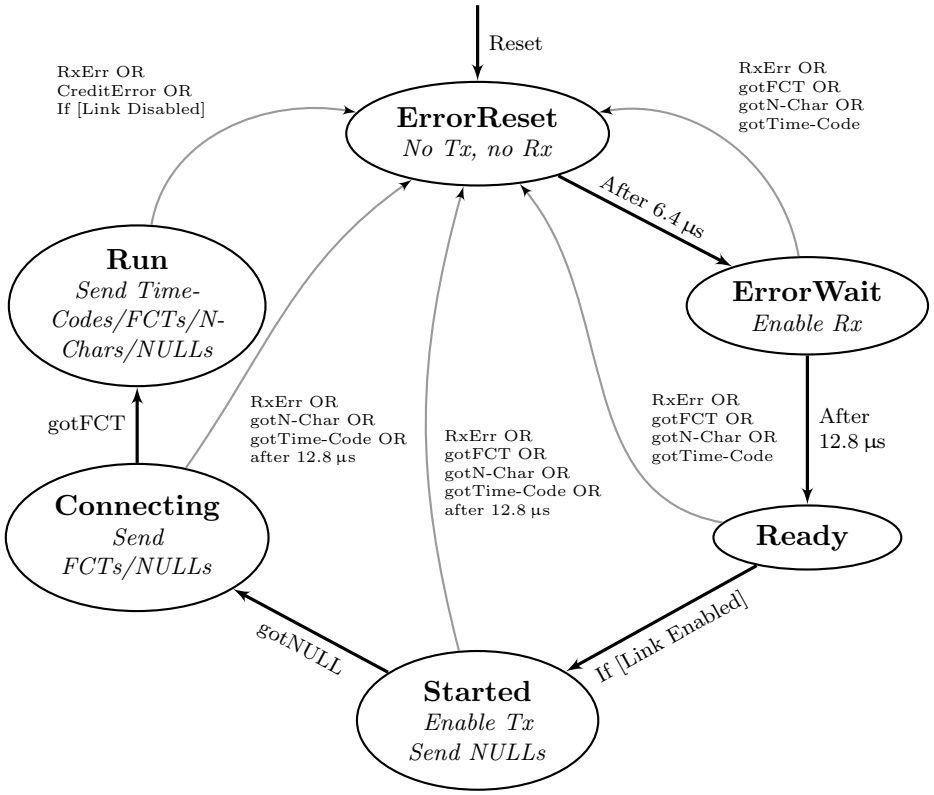


Figure 4.6: SpaceWire Link Interface State Machine. In normal operation, the link interface follows the thick arrows into the Run *state. On error it transitions back to the* ErrorReset *state, from where it will restart the link establishing sequence until it reaches the* Run *state. See section 4.2 for details. (adapted from Fig 20, page 60 in ECSS-E-ST-50-12C[7])*

packet. When an error occurs, the packet needs to be closed either with an EOP or EEP before sending new data characters.

In a network setting, the first data characters are to be interpreted as destination addresses, and thus stripped one after the next as the packet passes through the network.

4.2 Establishing a SpaceWire Link

Figure 4.6 displays a typical state machine of a SpaceWire link interface component such as the one used in this work. The following sequence of events lead to a successful SpaceWire link.

1. The link interface component starts out in the *ErrorReset* state. This state is also entered on every reset of the component, and whenever an error in the transmission occurs.

   In the *ErrorReset* state both the receiver *Rx* and the transmitter *Tx* are disabled.

2. After 6.4 µs the link component will transition to the *ErrorWait* state where the receiver is enabled. The component will stay in this state for 12.8 µs, and then enter the *Ready* state.

   These timeouts are used for ensuring a successful "exchange of silence", explained in section 4.2.3, in case an error occurred.

3. The specific SpaceWire link component used in this work publishes multiple input signals to give a fine-grained control on the link connecting behavior. It may be configured as "link enabled", "link disabled" or "automatic". In the simplest case, the component is enabled and will simply transition to the *Started* state and start the connection.

   Alternatively, the link may be disabled. In this case, the component will stay in the *Ready* state until the configuration changes or an error occurrs.

   In automatic mode, the link will only transition to the *Started* state when it detects that a connection attempt is made by the other side of the link. This is the case when the receiver detects a NULL-character.

   Automatic mode is what is used in the interface controller. That way it is up to the DPDPA in the rest of the spacecraft to enable or disable the nominal interface controller or the redundant one.

4. In the *Started* state, the link interface component enables the transmitter and sends out a continuous stream of NULL-characters to inform the other side of its existence.

5. Each side of the link can now detect a NULL-character from the opposite side. This tells each link component that the link is working and they can each enter the *Connecting* state.

6. Now the link is essentially established, and the two ends of the link inform each other how much room they have for receiving characters. This is done via flow control tokens (FCT) as described in the next section.

7. Upon detection of at least one FCT, each side can finally enter the *Run* state for normal operation.

4.2.1 Flow Control

The SpaceWire protocol ensures proper flow control of data characters. That is, each side of a SpaceWire link needs to keep track of how much space the other side has left for receiving characters. The maximum is to have room for 56 bytes of data, the minimum is 8 bytes. The maximum ensures that a 6-bit number is sufficient for keeping track. The minimum comes from the fact that each sending of an FCT implies that the sending side has room for receiving 8 more bytes. In this way, each side of the link knows exactly how many normal characters it is allowed to send across the link before receiving another FCT.

A *CreditError* occurrs when either too many normal characters or more than 7 FCTs corresponding to the maximum of 56 bytes have been received. If a *CreditError* occurrs, the link needs to be re-established.

4.2.2 The Run State

The *Run* state is the state of normal operation. In this state all kinds of characters can be sent from NULL-characters, normal characters, flow control tokens, and times-codes. These characters are sent with different priorities to ensure a smooth operation, as follows.

- NULL-characters are sent with the lowest priority. They are always sent if nothing else is pending in order to keep the link active and prevent the other side from detecting an exchange of silence.
- Of slightly higher priority are normal characters. These represent the data requested by the application. If the application indicates one should be sent, the link component waits for the current character to finish being sent and then immediately afterwards sends the normal data character.
- The second highest priority goes to flow control tokens. These must have a higher priority than normal characters, so that the receiving line is not drowned out by the sending line.
- In the *Run* state, time-codes are sent with the highest priority. They are not buffered and so do not need any form of flow control.

In any case, before sending a character, the link component waits for the current character to finish.

Initially, the clock frequency on the link must be $10 \pm 1\,\mathrm{MHz}$. During the *Run* state, the component may change to a different speed, where the lowest

is 2 MHz, and the highest is determined by skew, jitter, and the capabilities of the two ends of the link. The minimum speed is set since it counts as a disconnect error when neither the data nor the strobe signal changes within 850 ns. The SpaceWire core used in this work adjusts the clock speed automatically once in the *Run* state.

4.2.3 Link-Level Error Notification: The Exchange of Silence

There are three kinds of link-level errors that result in an "exchange of silence", although by some countings these are five errors:

- An *RxError* occurrs when either a disconnect was detected, a parity error occurred, or an escape error occurred.
  - A disconnect error is detected when there is a transition on neither the data nor the strobe signal for at least 850 ns.
  - A parity error occurrs when the number of bits in the parity coverage is not odd.
  - An escape error occurrs whenever an ESC character is followed by an invalid character, that is, it is neither followed by an FCT resulting in a NULL nor by a data character resulting in a time-code.
- A *character sequence error* occurrs whenever an unexpected character is received during link initialization, for instance, when an FCT is received before a NULL has been sent.
- Also, *CreditError*s as described in the flow control section 4.2.1 belong to the category of link level errors.

The two ends of the link notify each other of such link errors by an "exchange of silence". The side of the link detecting the error transitions to the *ErrorReset* state, thus causing a disconnect error on the other end of the link. Both sides then cycle through the link initialization procedure, during the first 19.2 µs of which no signal is transmitted.

Should one end have not correctly detected the disconnect, then the other side will cycle between the *ErrorReset* and *ErrorWait* states during which it transmits no signal. This exchange of silence ensures that a disconnect error is certainly detected on both ends.

Table 4.1: Transmitter and receiver host data interface encoding. (adapted from Table 7, page 54 in ECSS-E-ST-50-12C[7])

Control flag	Data bits (MSB ... LSB)	Meaning
0	xxxxxxxx	8-bit data
1	xxxxxxx0 (use 00000000)	EOP
1	xxxxxxx1 (use 00000001)	EEP

4.3 Application Side Interface

Although the standard does not mandate that implementations of the SpaceWire protocol follow the same application side interface specification, the recommendation summarized here is followed by at least two SpaceWire cores, including the one used for Simbol-X.

Only normal characters (data, EOP, and EEP characters) are exchanged with the application. Such characters between the link interface and the host application contain 9 bits. The first bit is the control flag, as indicated in Table 4.1. When the control flag is 1, the bits marked 'x' are ignored, but should be set to 0. If the control flag is 0, the other 8 bits contain the data. Also, link-level errors are reported via at least one signal, `RxError`.

4.4 Conclusion

SpaceWire is an easy to use high-speed point-to-point serial communication link with a low resource usage. A SpaceWire VHDL component synthesized completely inside an FPGA is quick to setup, allowing the developer to concentrate on other issues.

SpaceWire has been explicitly designed to be used in space applications, featuring very good immunity to electromagnetic interference. Speeds of 2 Mbit/s up to 400 Mbit/s are possible.

The overhead introduced by the SpaceWire protocol is 2 bits per byte. At a signaling frequency of 20 MHz as envisioned for Simbol-X, this allows a maximum data rate of 2 MB/s, well above the expected maximum of about half a megabyte per second.

As a standard, the SpaceWire protocol provides for seamless integration of satellite components developed around the world. The fairly clean application side interface allows a SpaceWire VHDL core to be reused in many components as well as in future missions.

Chapter 5

The SpaceWire-to-USB Converter SWitty

In this chapter SWitty is described. SWitty is a SpaceWire-to-USB converter to easily connect the interface controller or any other SpaceWire cable to a computer. The main component is a VHDL USB core that has a much more general application as a cheap method for transferring large amounts of data between hardware and software.

Before introducing SWitty, a different setup using a custom PCI card as a SpaceWire port to the computer shall be described. The problems associated with that setup is what gave rise to SWitty.

5.1 Test Setup with a SpaceWire PCI Card

For testing, the interface controller has a component to simulate data expected from the event pre-processor. To test the functioning of the interface

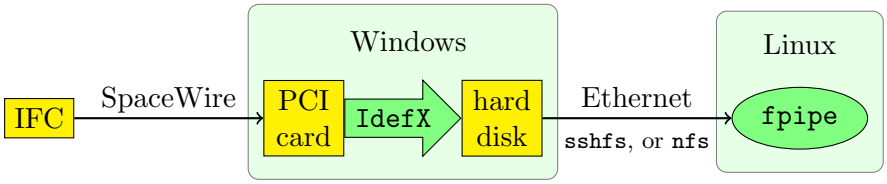


Figure 5.1: Test setup with SpaceWire PCI card. Randomly generated data from the IFC is transferred via a SpaceWire cable to a PCI card in a Windows computer. A special software, `IdefX`*, is used to save the data to disk, where it is transferred either via an* `sshfs` *or an* `nfs` *server to a Linux computer running the* `fpipe` *analysis software.*

controller, a test setup as shown in Figure 5.1 was used originally, where the interface controller transfers the simulated data via a SpaceWire cable to a SpaceWire PCI card in a Windows computer running specialized software that saves the data to disk. From there the data are sent via Ethernet to a Linux computer for analysis by the `fpipe` software.

Unfortunately, this setup has some major drawbacks. Foremost, special care needs to be taken to avoid data loss and corruption. Data that arrives on the SpaceWire port of the PCI card gets written into a direct memory access (DMA) ring buffer that is read and saved to disk by the special-purpose software `IdefX`.

The software does this by polling the ring buffer at regular intervals of at most once every millisecond. Should the software for some reason skip even a single polling interval, then a buffer overrun can occur where the current data are overwritten by new data coming in over the SpaceWire.

In theory, data rates of 8 MB/s are supported by the DMA-to-hard disk system compared to the 5 MB/s that are possible with the SpaceWire link. In practice, however, tests have shown that even at data rates on the order of 2 MB/s lead to an occasional buffer overrun.

To reduce the likelihood of buffer overruns, the `IdefX` software runs at the highest possible priority. However, overruns cannot be avoided completely this way when the data is simultaneously exported via either an NFS or SSH server over Ethernet to the Linux computer. The only workaround is to throttle the data transmission rate from within the interface controller.

The correct way to solve this would be to include flow control with the DMA buffer. However, the solution that was chosen for the lab setup completely sidesteps the SpaceWire PCI card, and makes use of the USB instead, resulting in a few other advantages elaborated on below. The result is that flow control is active along the entire chain from the interface controller to the analysis software.

The data loss problem at high transmission speeds is certainly the most difficult to work around, but there is another issue of usability. There is no direct way of automatically sending large amounts of data from the computer over the SpaceWire link *to* the detector. If, for instance, a new offset map was to be uploaded, then the experimenter would need to use some program to generate the offset map, and then he would have no choice but to convert the result into a file that `IdefX` can read, and, finally, turn to `IdefX` to send it off.

This is not very complicated, yet it adds another annoying and unnecessary level of complexity that cannot be solved easily with the PCI card and requires special care from the experimenter.

5.2 SpaceWire interfacing teletype

The solution offered here, called SWitty, is a SpaceWire-to-USB converter addressing the concerns with the PCI card. It makes use of operating system dependent interfaces, and should run on any system supporting the USB and specifically the *USB communications device class abstract control model* (USB CDC ACM) as standardized by the USB implementors forum at `http://www.usb.org`.

5.2.1 Software Interface and Features

SWitty is short for *SpaceWire interfacing teletype*. It is a USB device adhering to the USB CDC ACM standard, on one side, and a SpaceWire port on the other side. Flow control is built into the standard.

On Linux, the `cdc_acm` driver module creates the device special file `/dev/ttyACM0` when the device is attached on the USB. From the view of the software application, the device special file is used as an ordinary file, except that what is written to it is not necessarily what is read from it.

Sending data from the host computer to the detector is as simple as writing to `/dev/ttyACM0`.

Receiving data from the detector over the SpaceWire link is as simple as reading from `/dev/ttyACM0`.

Other operating systems have different interfaces for SWitty. In this work, Linux has been used for its rather strong USB debugging facilities and the fact that the `fpipe` analysis software runs on it.

The device special file `/dev/ttyACM0` must be configured via the command

```
$ stty -F /dev/ttyACM0 raw -echo -echoe
```

This disables any interpretation and echoing back of the bytes in the data stream. It is necessary to run this command on every re-load of the USB CDC ACM device driver, such as a reboot.

The reason this needs to be run is that, traditionally, teletypes were used as terminals for users to type in commands. These would be commanded by specially interpreted characters such as *carriage return* and *new line*. For SWitty these are turned off by the `raw` option.

Also, the user would ordinarily want to see the result of her typing being echoed back to the screen of her teletype. A software emulation of this behavior is still used for the console of every Linux computer, which is why the `echo` and `echoe` options are enabled by default and must be turned off.

Apart from solving the flow control and usability issues of the PCI card, SWitty inherits some limitations from the USB so that an exact one-to-one and onto relation between SpaceWire and the USB cannot be achieved. However, these complications are not very severe.

First, care needs to be taken not to overload the USB by attaching too many devices to the same root hub. This would result in very slow transmission speeds. Since most computers come equipped with more than one root hub, this issue is not much of a concern.

Second, SpaceWire time-codes cannot be sent via SWitty easily. The reason behind this is due to the USB's master-slave architecture that introduces a high latency. However, a partial solution to this problem could be to exploit *start-of-frame* markers as sent by the USB every millisecond, and to publish this as a time-code over SpaceWire. For Simbol-X, a different time synchronization mechanism is envisioned, so this idea has not been developed further. It is mentioned here for future reference.

On the other hand, SWitty also offers a few advantages over the SpaceWire PCI card. The USB is cheap and ubiquitous. All hardware is already available on most any computer and many FPGA development boards come equipped with a UTMI, a USB 2.0 interface chip that eases USB development.

The standard device class allows SWitty to be used with just about any operating system supporting the USB communications device class abstract control model, although tests were done exclusively on Linux.

The driving force behind the development of SWitty, however, comes from its usability as a generic way to transfer large amounts of data from an FPGA to a computer and vice versa. The advantage over commercially available solutions is the increased data rate by a factor of 10-15. Theoretically, 40 MB/s should be possible, although this places extreme demands on the cable quality, and is only possible if the hardware FPGA design is specially created for such data rates. In reality, data rates of about 10 MB/s are well achievable.

5.2.2 The Universal Serial Bus

Starting in the 1990s, the *Universal Serial Bus* (USB) has been developed by a collaboration of major computer manufacturers. It is primarily designed to be used as a plug-and-play desktop peripheral connection standard. While the details of the USB topology and protocol are discussed in Appendix A, some basic terms shall be defined in the following points.

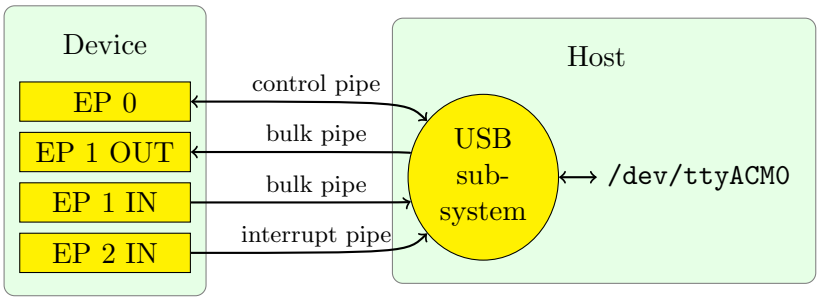


Figure 5.2: In the high-level communication model of the USB, pipes are multiplexed over the low-level USB cable, so that the USB subsystem of the operating system essentially talks directly with an endpoint (EP). The example here shows the endpoints of a CDC ACM device that is published by the Linux operating system as a device special file `/dev/ttyACM0`.

- A USB cable has four wires, two that carry a differential logic signal, and two as a 5 V power supply.
- Every device on the USB is assigned a unique 7-bit *device address*.
- The USB distinguishes *low-speed*, *full-speed*, and *high-speed* devices. A device is first attached as a full-speed device. By initiating a *device chirp*, it can advertise itself as a high-speed device when first connected.
- A device consists of a set of *endpoints*. An endpoint is viewed as a FIFO that either accepts or sends data. It is identified by the *endpoint address* and the direction of data flow.
- The high-level communication with an endpoint is via a *pipe*. The USB subsystem of the operating system multiplexes the different pipes over the USB cable, so that a specific driver does not need to take care of the low-level USB protocol, see Figure 5.2.
- A pipe is unidirectional. That is, an OUT endpoint can only accept data from of the host, and an IN endpoint can only send data from the device to the host.

  Only the pipe associated with endpoint zero is bidirectional. It is used for control transfers.
- Data is exchanged over a pipe via *transactions* initiated by the host. The device must wait for the host before sending any data. Due to this host-centricism, USB latency is on the order of milliseconds.
- There are four kinds of pipes: control pipes, bulk pipes, interrupt pipes, and isochronous pipes. In this work only control pipes and bulk

pipes are used. Control pipes are used for determining the type of device. Bulk pipes are used to transfer large amounts of data reliably.

- The bulk transfer protocol includes flow control and data integrity checking.

- The USB employs the notion of standardized *device classes*. The most common are the *USB mass storage device* class for memory sticks and the *human interface device* class for mice and keyboards. In this work, the *USB communications device class abstract control model* (USB CDC ACM) is used to emulate a serial-to-USB converter.

- On Linux, the software interface for a USB CDC ACM device, and by extension for SWitty, is very simple. Sending data to the device is done by writing to the device special file `/dev/ttyACM0`, while receiving is done by reading from that file.

- Theoretically, speeds up to 40 MB/s are possible, but interference and electronics design limit the practical rate to about 10 MB/s.

5.2.3 SWitty Design

The original implementation of the USB communications class VHDL component available at [9] is only capable of USB 1.1 speeds. That original has been heavily changed in this work to allow for USB 2.0 high-speed transactions. Figure 5.3 shows a schematic of SWitty. The components fulfill the following tasks.

- The component `serswi` connects the SpaceWire component on the left with the USB system on the right.

- The component `usb_data` implements the USB CDC ACM data interface. It contains the bulk IN and bulk OUT endpoints used for the data transmission.

  The bulk IN endpoint is essentially a 512-byte FIFO with some control logic. The USB specification mandates that a bulk endpoint has a size of 512 bytes.

  The bulk OUT endpoint is implemented as a 1024-byte memory. A FIFO cannot be used, since a data packet needs to be re-sent should an error occur in the transmission. For efficiency reasons, the memory is twice as large as necessary. This allows another packet to be saved in memory while the first packet is being transmitted.

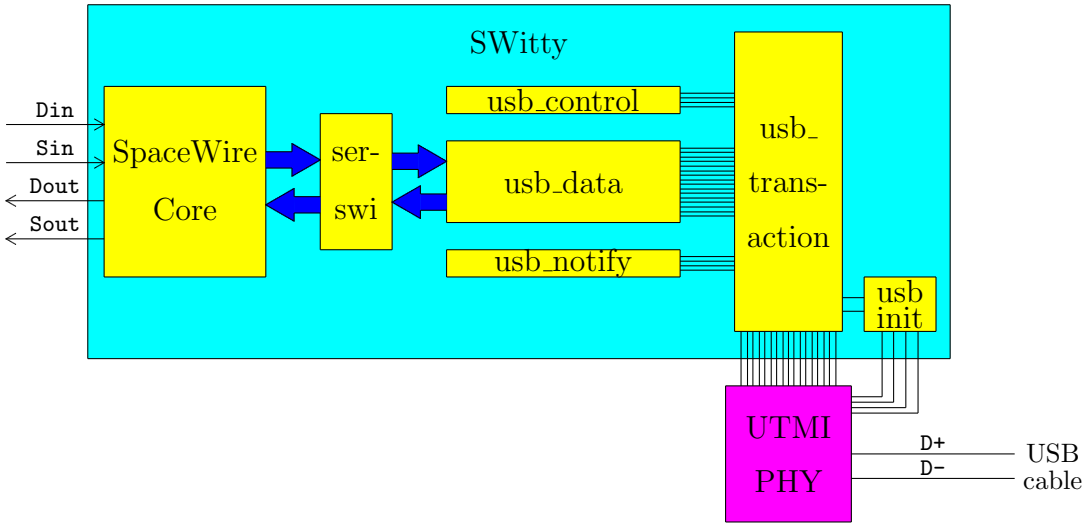


Figure 5.3: This diagram shows a schematic of SWitty. At the bottom right are the USB cable signals drawn connected to the UTMI. The UTMI is connected with the component `usb_transaction`*, which handles all aspects of a USB transaction. The UTMI and* `usb_transaction` *are controlled by* `usb_init`*, which monitors the USB* `D+` *and* `D-` *lines for reset signaling. The components* `usb_control`*,* `usb_data`*, and* `usb_notify` *contain the endpoints.* `usb_serswi` *is a serial-to-SpaceWire converter to connect the USB system with the SpaceWire core.*

- The `usb_control` component is the implementation of endpoint zero. Its main purpose is to inform the host that SWitty is a USB CDC ACM device. This is done by transmitting *device* and *configuration descriptors* when the device is first attached on the USB.
- The `usb_notify` component implements an interrupt IN endpoint as mandated by the CDC ACM device class. As the endpoint is not used for SWitty, it just sets some constants for `usb_transaction` that result in no data being sent whenever the host sends a request to it.
- The component `usb_transaction` checks for the correct device address, multiplexes host requests to the correct endpoint, checks the data integrity, and sends handshake packets at the end of a transaction as requested by endpoints. In short, it provides a convenient transaction-oriented interface for endpoints, taking care of the individual packet details.
- The UTMI chip provides the low-level electrical interface for the USB cable lines. Its interface is described in the appendix in section A.1.3.

Table 5.1: SpaceWire encoding over 8-bit byte stream.

Byte Sequence	Interpretation	SpaceWire Character
`0x00 0x00`	treated as single `0x00`	
`0x00 0x01`	`EOP`	`"100000000"`
`0x00 0x02`	`EEP`	`"100000001"`
`0x00 0xff`	null data character	`"000000000"`
`0x00` others	disallowed, signifies error	`"100000001"`
others	data byte with given value	`"0XXXXXXXX"`

- Finally, the `usb_init` component monitors the USB cable through the UTMI for resets by the host, and initiates the device chirp to advertise itself as a high-speed device.

5.2.4 SpaceWire Character 8-bit Encoding

Although a SpaceWire data character carries 8-bits of arbitrary data, two additional characters, the EOP and the EEP need to be escaped before they can be sent over the 8-bit data stream of the USB.

The encoding is accomplished by designating the character `0x00` as the escape character in the 8-bit byte stream. Any character immediately following the escape character is specially interpreted according to the rules in Table 5.1. The `EOP` and `EEP` characters are encoded as the escaped `0x01` and `0x02`, respectively. As the data byte consisting of all zeros is the escape character, it, too must be escaped. It is encoded as the escaped `0xff`. Other escaped bytes signify an error. They are currently treated as an `EEP`, even though technically the `EEP` should be issued only at the next `EOP` or `EEP`. Non-escaped characters are data characters in the 9-bit SpaceWire representation.

The escaped escape character collapses to a single escape character. This provides for simpler and more robust decoding. To see this, imagine we had encoded the null data character as `"0x00 0x00"`, that is, as two consecutive escape characters. Now, consider the following snippet from a byte stream:

```
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x34 0x67
```

In order to decode the byte `0x01` correctly we would need to know whether the preceding `0x00` was meant to escape the `0x01`, or whether it was to be interpreted as a data character of all zeros in its own right. For the given byte sequence snippet this cannot be determined without careful tracking of the preceding history, while for the actual encoding scheme used it is clear

that this sequence decodes to the three SpaceWire characters `"EOP 0x34 0x67"`.

5.3 Conclusion

SWitty is a SpaceWire-to-USB converter with a simple and elegant software interface on Linux. Although a one-to-one and onto relation cannot be achieved with the USB, SWitty solves all problems and annoyances that occurred with a custom SpaceWire PCI card.

Using a standard USB device class, SWitty should be able to be used with most any computer operating system that supports the USB, although only Linux has been used thus far. The software interface differs between operating systems to accommodate for their respective philosophies. On Linux, sending and receiving is achieved by simply writing and reading the device special file `/dev/ttyACM0`.

Although SWitty was introduced to solve the inconveniences with the SpaceWire PCI card, the core USB component resulting from that has a much more general use as a way to transfer large amounts of data between software and hardware. It is this general applicability that motivates its further development.

Conclusion and Outlook

The development of the interface controller for the low energy detector of Simbol-X has resulted in a simple and highly extensible design. It has been shown with various test setups that the interface controller and SpaceWire link are capable of easily fulfilling the requirements for the detector connection with the telemetry system.

The SpaceWire protocol has been found to be extremely reliable and suited for space applications. It is already used in recent satellite missions providing them with a fast, reliable and standardized communication protocol as described in this work. Although financial problems have prevented the start of phase B for Simbol-X, the emerging SpaceWire protocol will be used in many future space missions such as the *High Timing Resolution Spectrometer* onboard the planned *International X-ray Observatory* (IXO).

The SpaceWire-to-USB converter that is called SWitty and was created as part of this work has greatly facilitated testing the interface controller. As a general method for cheaply transferring large amounts of data between hardware and software, the main USB component of SWitty is planned to be used for testing other components of the detector electronics, in particular the analog-to-digital converter that digitalizes the raw pixel values from the detector matrix.

In the coming months it is planned to integrate the low energy and high energy detectors originally foreseen for Simbol-X in the science verification model being setup in Tübingen. The interface controller will play a major role in that effort. It will be interesting to test the interface controller and SpaceWire connections in context and to gather the performance characteristics of the combined high and low energy detectors.

Bibliography

[1] B. W. Carroll and D. A. Ostlie. *An Introduction to Modern Astrophysics*. Addison-Wesley Publishing Company, Inc., 1996.

[2] D. Clowe, M. Bradač, A. H. Gonzalez, M. Markevitch, S. W. Randall, C. Jones, and D. Zaritsky. A Direct Empirical Proof of the Existence of Dark Matter. *The Astrophysical Journal Letters*, 648:L109–L113, September 2006.

[3] S. Corbel. GRS1915, November 2009. `http://irfu.cea.fr/Sap/Phys/Sap/Actualites/CORBEL/corbel_gb.shtml`.

[4] F. Cordero. Simbol-X Detector Payload SpaceWire Utilisation Requirements; SX-SP-1-21-CESR; Draft Edition 2, Revision 0, 22/10/2009, October 2009.

[5] Intel Corporation. USB 2.0 Transceiver Macrocell Interface (UTMI) Specification, March 2001. `http://www.intel.com/technology/usb/download/2_0_xcvr_macrocell_1_05.pdf`.

[6] NASA/CXC/CfA/ R.Kraft et al. (X-ray); MPIfR/ESO/APEX/ A.Weiss et al. (Submillimeter); ESO/WFI (Optical). Centaurus A, November 2009. `http://chandra.harvard.edu/photo/2009/cena/`.

[7] European Cooperation for Space Standardization. SpaceWire – links, nodes, routers and networks, July 2008. ECSS-E-ST-50-12C, `http://www.ecss.nl/forums/ecss/_templates/default.htm?target=http://www.ecss.nl/forums/ecss/dispatch.cgi/standards/docProfile/100302/d20060808084754/No/t100302.htm`.

[8] USB Implementors Forum. Universal Serial Bus Specification, April 2000. `http://www.usb.org/developers/docs/usb_20_052709.zip`.

[9] Jori. USB data transfer in VHDL, April 2009. `http://www.xs4all.nl/~rjoris/fpga/usb.html`.

[10] NASA/CXC/SAO. Chandra image of the Crab, November 2009. `http://chandra.harvard.edu/photo/2008/crab/`.

[11] NASA/STScI. Hubble Space Telescope Crab mosaic, November 2009. http://hubblesite.org/newscenter/archive/releases/2005/37/image/a/format/small_web/.

[12] X-ray optics. Curved mirror optics, November 2009. http://www.x-ray-optics.de/index.php?option=com_content&view=article&id=59&Itemid=71&lang=en#Wolter_optics.

[13] J. A. Peacock. *Cosmological Physics*. Cambridge University Press, January 1999.

[14] F. Pinsard and C. Cara. High Resolution Time Synchronization over SpaceWire Links. 1-4244-1488-1/08©2008 IEEE, IEEEAC paper#1158, Version 2, Updated 2007:12:07, December 2007.

[15] P. Schneider. *Extragalactic Astronomy and Cosmology*. Berlin: Springer, 2006.

Appendix A

The Universal Serial Bus

This chapter is about the details of the Universal Serial Bus. It is used in SWitty, but the resulting component has a much more general application as a way to transfer large amounts of data between hardware and software.

The chapter is heavily based on the USB 2.0 specification[8]. A more general introduction is given there. The purpose of this chapter is to give a concise intermediate introduction to the USB, in particular to those aspects that are used in SWitty. For the design and usage of the component, see the chapter on SWitty, chapter 5.

A.1 USB Basics

The Universal Serial Bus (USB) has been developed during the 1990s by a consortium of hardware manufacturers and software companies, including Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips. Revision 0.7 of the specification was released on November 11, 1994, revision 1.0 on January 15, 1996, revision 1.1 on September 23, 1998, and revision 2.0 on April 27, 2000. More recently revision 3.0 was made available on November 12, 2008. The topic of this chapter is the USB 2.0 specification[8], as USB 3.0 hardware was not readily available as of this writing, September 2009. It is now promoted by the USB Implementors Forum at `http://www.usb.org`, a non-profit corporation founded by the companies involved in the USB's development.

The USB was developed to replace the myriad of incompatible connectors and differing protocols at the time by a single easy-to-use and safe standard for connecting peripheral devices to a computer. Apart from featuring plug-and-play, the USB's connectors are designed to be easily pluggable. They are designed such that it is impossible to create an illegal link.

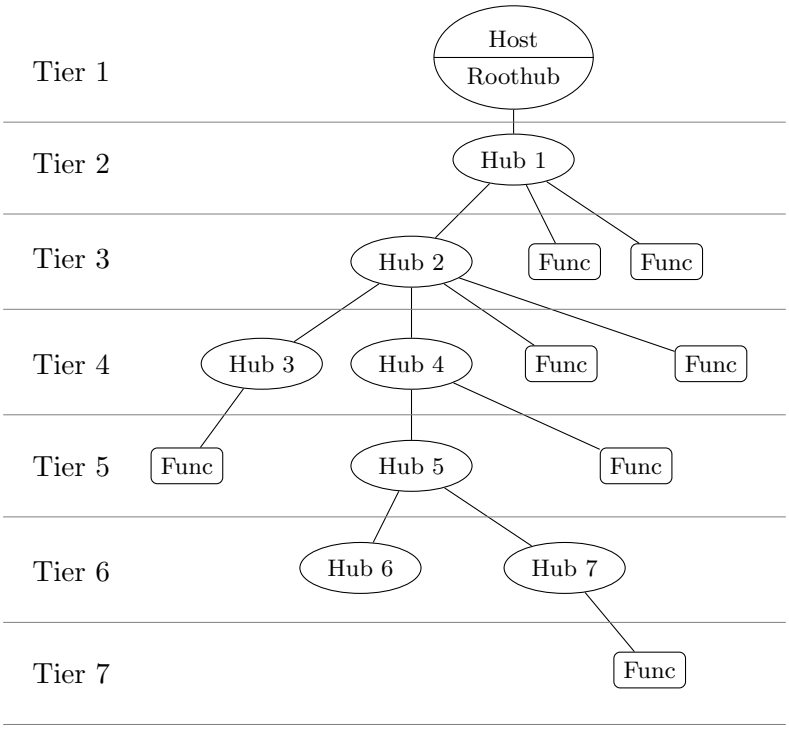


Figure A.1: This figure shows the USB tiered star topology. Each device is connected to a hub in the next higher tier. Hubs multiplex the connections to the port they are attached to. (adapted from Figure 4-1 of the USB 2.0 specification[8])

Initially, the USB supported 12 Mbit/s and 1.5 Mbit/s transfer rates, termed *full-speed* and *low-speed*. The purpose of *low-speed* devices was to allow for cheaper and lighter peripherals with thinner cables. With USB 2.0 the need for higher transfer speeds was addressed by introducing *high-speed* with a transfer rate of 480 Mbit/s. Low-speed, full-speed, and high-speed peripherals can all be connected with the same cables.

A competing standard is IEEE 1394, termed "FireWire". It emerged around the same time as the USB, but is more expensive and not quite as ubiquitous as the USB.

In the following, the basic setup and terminology of the USB is explained.

A.1.1 Topology

USB devices are arranged in a tiered star topology as shown in Figure A.1, where a hub is a device that provides USB ports for additional peripheral devices, called *functions*. As shown in the figure, there may be no more than

7 tiers. The electrical design constrains a USB cable to a maximum length of 5 m, or 3 m for low-speed devices. The last hub can only be part of a USB peripheral, so the maximum length that a USB peripheral can be from the host computer is 30 m. The USB was designed as a desktop peripheral system, so these constraints were deemed acceptable.

At the top is the root hub inside the host computer. Within a tier, a hub is the center star for the functions and hubs attached directly to it from the next tier. Two functions cannot be connected directly or indirectly with each other. The USB plugs are designed to make such illegal connections impossible.

A function is *downstream* of the hub it is connected to, the host is *upstream*. The port on a function is called a downstream port or an upstream facing port. Similarly, the downstream facing port on a hub is called the upstream port of the link between function and hub. In this view, commands are generated at the host, flowing downstream towards a function.

A function is viewed as consisting of several endpoints, physically corresponding to a FIFO each. An endpoint may either be an IN or an OUT endpoint depending on the direction of data flow as seen from the host. Only the endpoint zero may communicate bidirectionally. All others may only be IN or OUT. Thus, an IN endpoint can only send data to the host, while an OUT endpoint receives data from the host.

The USB is designed so that a software running on the host computer is effectively communicating directly with the endpoints in a function. Addressing the particular function in question is the task of the USB subsystem inside the operating system. Conversely, the function is responsible for only participating in transactions destined for itself.

Upon attachment, each function gets assigned a unique 7-bit address. Before assignment the function must respond to the null-address. Thus, a maximum total of 127 functions may be connected to any single root hub.

An endpoint is identified by a 4-bit endpoint number and the direction of data flow. Endpoint zero is bidirectional. Thus, per function there may be at most 15 IN and 15 OUT endpoints in addition to endpoint zero.

A set of endpoints is collected into an interface. There may be more than one interface in a function. More on that later in section A.2.4.

A.1.2 Physical Layer

A USB cable physically consists of the data lines `D+` and `D-`, a ground line, and a +5 V power line.

The two data lines can be in any of four states: SE0, K, J, and SE1. SE0 is short for "single ended zero", and indicates that there is no link activity. The "single ended one", SE1, is not allowed in a high-speed link.

The states K and J represent the logical 1 and 0, where the signal `D+` is high and `D-` is low for the K state, and vice versa for the J state. Together, `D+` and `D-` provide a single differential signal for data transfer.

Only one side of a USB link can drive the `D+` and `D-` signals at a time. The protocol defines which end of the link is currently driving, and which is receiving.

To provide a basic level of error detection and clock synchronization during transfers, USB data streams employ bit stuffing and NRZI encoding. Packets are started with 32 sync bits, used to synchronize the receiving clock with the sender. The end of a USB packet is indicated with a 2-bit end-of-packet marker. These low-level mechanisms are not the topic of this work, but are defined in chapters 6 and 7 of the USB 2.0 specification[8].

A.1.3 USB 2.0 Transceiver Macrocell Interface

Intel developed in 2001 the *USB 2.0 Transceiver Macrocell Interface*, or UTMI[5] for short. The most important reason being that only few FPGAs can run at 480 MHz required for high-speed devices. Thus, the most important task of a chip implementing the UTMI is to convert the high-speed frequency serial data to 60 MHz parallel data.

To accomplish this, a UTM must revert bit stuffing and NRZI encoding. In addition to that, a UTM takes care of sync bits and end-of-packet markers, and allows setting the termination resistors for high-speed and full-speed operation.

Receiving a packet follows the following logic:

- When the UTM detects a valid sequence of sync bits on the bus, it raises the signal `RXACTIVE`.
- When the UTM has successfully decoded and deserialized a byte, it raises the signal `RXVALID`, and publishes the data on the bus `RXDATA`.
- On the next clock cycle the UTM puts the next byte on `RXDATA`, or else `RXVALID` is lowered. The application must react immediately.
- When an end-of-packet marker or an error has been detected, the UTM pulls `RXACTIVE` low.

Sending a packet works as follows.

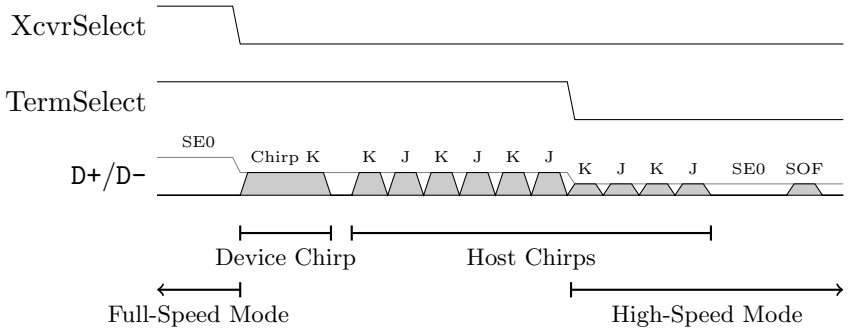


Figure A.2: USB High-Speed Detection Handshake. After the reset via an extended SE0, the device chirp lasts for at least 1 ms*, and at most* 7 ms*. The host responds with alternating K and J states each lasting at least* 40 µs *and at most* 60 µs*. After six K and J states, the function enables its high-speed termination resistors. The host drives a SE0 for* 100 µs *to* 500 µs *before entering normal high-speed operation with the first start-of-frame packet (SOF). (Figure 25 in the UTMI specification [5])*

- The application puts the data on the parallel bus `TXDATA` of the UTM and raises the signal `TXVALID`.
- The UTM then sends the sync bits and the data byte, and then raises the signal `TXREADY` to inform the application.
- The application must then put the next byte on `TXDATA`, or lower `TXVALID` to indicate that an end-of-packet marker should be sent.

The `D+/D-` signals are exported with the 2-bit bus `LineState`.

With the signals `XcvrSelect` and `TermSelect` the termination resistors on the `D+/D-` lines are chosen to select between high-speed and full-speed operation.

A.1.4 High Speed Detection Handshake

All USB peripherals initially connect as full-speed devices. That is, `XcvrSelect` and `TermSelect` are set high for full-speed operation. With the high-speed detection handshake, it is determined if both device and host are high-speed capable.

The high-speed detection handshake is initiated after a reset. When the function detects an SE0 on the `D+/D-` lines for at least 3 ms, then this indicates a reset, and the function initiates the high-speed detection handshake, as shown in Figure A.2.

To advertise itself as a high-speed capable device, the function drives a K state on the `D+/D-` lines for between 1 ms and 7 ms. For this, the transceiver must be set to high-speed mode by putting `XcvrSelect` low. The high-speed signaling level is lower than that of full-speed, as schematically shown in Figure A.2. This is called the *device chirp*, and it is below the detection threshold of full- and low-speed only hubs.

If the hub is high-speed capable, it responds with the *host chirp* by driving alternating K and J states. When the function has detected six K's and J's, it switches to high-speed termination resistors by pulling `TermSelect` low on the UTMI. Now high-speed operation is enabled. The hub may drive a few more K and J chirps, and then drive SE0 for a short time to indicate the end of the high-speed detection handshake.

The device is prevented from detecting a reset by the continuous sending of start-of-frame (SOF) packets as discussed below in section A.2.2.

A.2 USB Communication Protocol

While the last section was mostly concerned with the electrical level of the USB, this section explains how the logical interface works by defining what a packets are and how they are exchanged. It is essential to keep in mind that the USB is a host centric protocol. That is, all transfers are initiated by the host.

A.2.1 Packets

A USB packet is started with a sequence of sync bits, either 8 bits long for full-speed communication, or 32 bits for high-speed. It is ended with a 2-bit end-of-packet marker. These are identified and stripped by the UTMI, and will thus be ignored in the following discussion.

The first bit transmitted via the UTMI is the packet identification byte, the PID. Possible values for the four low bits are shown in Table A.1. The high bits are set to the bitwise complement of the low bits, e.g.

$\overline{p_3}$	$\overline{p_2}$	$\overline{p_1}$	$\overline{p_0}$	p_3	p_2	p_1	p_0

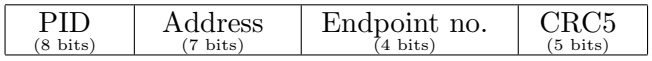
If the received PID is not mirrored in the high bits, then this indicates an error and the packet must be ignored.

The PID decides which of the following three types of packet is being sent.

Table A.1: Lower PID bits. The upper PID bits are the bitwise complements, as described in the text. Only those shown are needed for a USB 2.0 function. The complete list of PID types is in Table 8-1 of the USB 2.0 specification[8]. It includes two more data PIDs for isochronous transfers, three more special PIDs for hubs, and one reserved for future use.

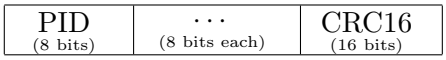
Type	PID	Value	Description
Token	OUT	0001	Initiates OUT transaction
	IN	1001	Initiates IN transaction
	SOF	0101	Start-Of-Frame
	SETUP	1101	Initiates OUT transaction on control pipe
Data	DATA0	0011	Data packet with toggle bit 0
	DATA1	1011	Data packet with toggle bit 1
Handshake	ACK	0010	Acknowledge error-free delivery
	NAK	1010	Temporarily cannot receive or accept data
	STALL	1110	Endpoint requires host interaction
	NYET	0110	Endpoint is not yet ready
Special	PING	0100	Check if bulk OUT endpoint is ready to accept data

- Token packets other than SOF have the following four fields for a total of three bytes, where the CRC5 field covers only the address and the endpoint number fields.



  The CRC is checked by calculating the residual of the last two bytes.

- Data packets have the following form with the CRC16 covering the data bytes but not the PID.



- Finally, handshake packets and the PING special packet consist of only the PID.

In this discussion the sync bits and the end-of-packet markers have been left out, since the UTMI does not pass these on.

A.2.2 USB Frames and Microframes

The host must send Start-Of-Frame (SOF) packets at the start of every frame for full-speed devices, or at the start of every microframe for high-speed devices.

A frame lasts 1 millisecond, a microframe 125 microseconds.

SOF packets have the following formatting, where the time is an 11-bit number that is incremented at the start of every frame.

For high-speed devices, 8 identical SOF packets are sent per frame, each indicating the start of a microframe.

SOF packets serve two purposes. For one, it insures link activity, so the function does not suspend or reset. Additionally, the SOF can be used as a timing source for the function, when the function, for example, operates as an audio device.

In the present context, SOF packets are ignored, but could be used to send a time character over the SpaceWire link.[1]

A.2.3 Transactions and Transfer Types

Transferring data between the function and the host is done via transactions on the USB. A transaction most generally consists of the exchange of 3 packets, a token packet, a data packet, and a handshake packet, as shown below.

The USB is a host-centric protocol. Only the host can send token packets and thus initiate a transaction. The PID of the token packet identifies the type and destination of the transaction as per Table A.1.

For OUT and SETUP transactions, the data packet is sent by the host to the function. For IN transactions, the function needs to respond by sending a data packet to the host.

The handshake packet is sent by that end of the link that did not send the data packet. It is used to acknowledge the success of a transaction.

[1] SOF packets and Frames are more closely described in section 8.4.3 of the USB 2.0 specification[8].

The USB distinguishes four types of transfers. These are control transfers, bulk transfers, interrupt transfers, and isochronous transfers. A transfer of data consists of one or more transactions.

- *Control transfers* are associated with endpoint zero. This type of transfer is the only one that is bidirectional. It is used for configuring and enumerating a device.
- *Bulk transfers* feature guaranteed data delivery. Together with control transfers they are the main topic of this chapter. Bulk transfers are generally used in mass storage devices. Data rates up to 40 MB/s can be achieved with this kind of transfer.
- *Interrupt transfers* are used to deliver data reliably with low latency, as is desirable for human interface devices such as a mouse or keyboard. Interrupt transfers can sustain a maximum data rate of 24 MB/s.
- *Isochronous transfers* guarantee bandwidth negotiated at attach time, but there is no guarantee of data delivery. The handshake packet is not part of an isochronous transfer. Isochronous transfers are useful for streaming audio or video where the timely arrival of the data is more important than its integrity.

The abstract communication of the host with an endpoint is called a *pipe*. A pipe may use any of the transfer methods described, resulting in a control pipe, a bulk pipe, an interrupt pipe, or an isochronous pipe.

A.2.4 Device Descriptors

A device is described by *descriptors*, defined in section 9.6 of the USB 2.0 specification. The hierarchy of descriptors is depicted in Figure A.3. At the top is the *device descriptor*. It includes general information about the device such as the vendor and product id, as well as the number of configuration descriptors.

A device may have one or more *configuration descriptors*, although typically there is only one. It is the host that chooses the configuration. The descriptor informs the host of the number of interfaces within the configuration and the maximum amount of power the device is going to drain from the bus.

A configuration may have multiple interfaces. An interface is a set of endpoints acting together to achieve a function. The *interface descriptor* includes the interface class id and the number of endpoints in this interface. The interface class is used by the host operating system to load the device driver for that specific class.

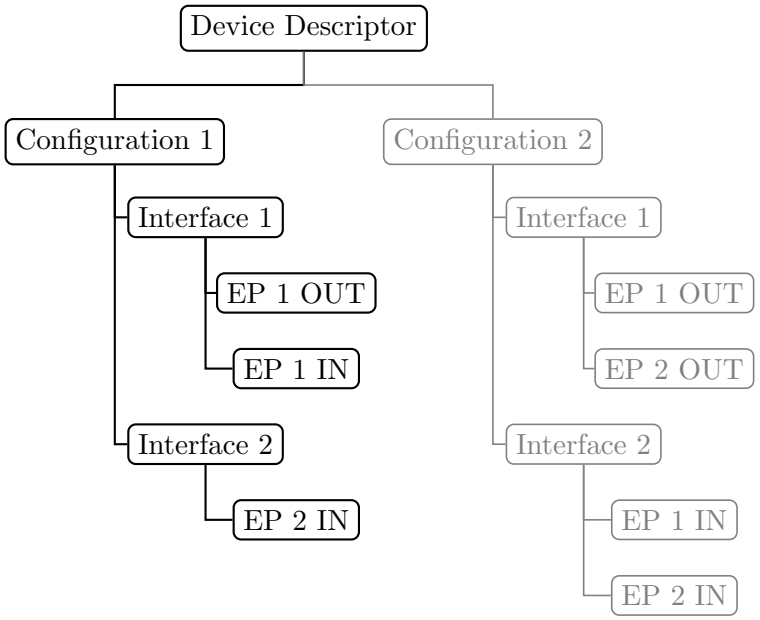


Figure A.3: A device is described by exactly one device descriptor. One out of many configurations can be chosen. Each configuration can consist of multiple interfaces. An endpoint is abbreviated as EP. It is identified by its endpoint number and the direction of data flow as seen from the host. Each interface can have as many endpoints as desired. Endpoint zero and functional descriptors are not shown.

Normally, a device driver in the host operating system binds to the endpoints of a single interface. Exceptions to this rule must specify the device class in the device descriptor. The communications device class for modems and Ethernet adapters used in this work is such an exception. In that case the endpoints from two interfaces act together as a single interface.

To describe an interface more fully, the USB allows for *functional descriptors*. They are specific to the interface class and, for the purposes of this work, are filled with some default values.

Finally, an *endpoint descriptor* specifies for each endpoint the address and direction, the transfer type, and the maximum packet size. The endpoint address and direction are encoded in a single byte, where the lower 4 bits are the endpoint address, and the highest bit is the direction, 0 for OUT and 1 for IN. The other bits must be set to 0. The transfer type of an endpoint is specified in the lowest 2 bits of a byte, as shown in the following table.

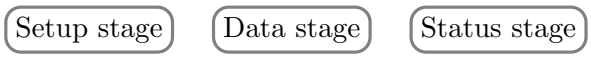
00 = Control
01 = Isochronous
10 = Bulk
11 = Interrupt

The other bits are reserved for isochronous endpoints, and must otherwise be set to zero.

The configuration, interface, functional, and endpoint descriptors are all transmitted together when the configuration descriptor is requested. The total length of these descriptors is specified in the configuration descriptor.

A.2.5 Control Transfers

A control transfer generally occurs in three stages, each stage forming a 3-packet transaction as described in section A.2.3.



In the setup stage, data is transferred from the host to the endpoint. The endpoint responds by sending data to the host during the data stage. In the status stage the host acknowledges the reception of the data.

The data stage may be left out for some setup transfers. During the status stage the endpoint then transfers data from the function to the host. No data in the status stage indicates success.

The data packet in the setup stage carries the following information.

bmRequestType (8 bits)	bRequest (8 bits)	wValue (16 bits)	wIndex (16 bits)	wLength (16 bits)

The *bmRequestType* and the *bRequest* fields determine what actions are taken by the function, and how the other fields are to be interpreted. Not all actions need to be implemented by a device. Table A.2 summarizes the bare minimum that is needed for a function's endpoint zero.

A.2.6 The `usbmon` Linux Kernel Module

The Linux kernel, developed by Linus Torvalds starting in 1991 and many others since, provides within the `sysfs` pseudo-filesystem the `debugfs`. It is mounted at `/sys/kernel/debug`, and allows kernel modules to export debugging information of arbitrary formatting to userspace applications.

Table A.2: Summary of important setup requests implemented in `usb_control.vhd`. *The SETUP packet column displays the values of the* bmRequestType, bRequest, wValue, wIndex, *and* wLength *fields in the setup stage of a control transfer in hexadecimal notation. The Request column displays the common name, and the function response is listed in the last column. The complete list is found in chapter 9 of the USB 2.0 specification[8].*

SETUP packet	Request	Function Response
00 05 00XX 0000 0000	SET ADDRESS	Set device address to XX. Return empty status stage.
80 06 0100 0000 ZZZZ	GET DESCRIPTOR	Return up to ZZZZ bytes of the device descriptor.
80 06 02YY 0000 ZZZZ	GET DESCRIPTOR	Return up to ZZZZ bytes of the configuration descriptor with index YY.
00 09 00NN 0000 0000	SET CONFIGURATION	Choose and set the configuration with index NN.

When loaded, the `usbmon` kernel module creates pseudo-files in `/sys/kernel/debug/usb/usbmon` in the most recent kernel version 2.6.31, or one directory higher on older kernels. The directory listing may look like this:

```
0s  0u  1s  1t  1u  2s  2t  2u  3s  3t  3u  4s  4t  4u
```

The numbers in the filenames indicate the root hub number, the letter the format. The `'0'`-files are not corresponding to a root hub, but instead display information from all root hub.

We are only concerned with `'u'` formatted files. They display the transactions that transfer data over the bus. Each line stands for a transaction. The first column in a line displays a time in microseconds since an arbitrary point in the past. The second column tells us whether the host is "submitting" (S) the transaction, or if the host is expecting a "callback" (C) from the device. The example below shows the traffic between the hub and the host when a device is first attached.

```
520006582 S Ci:2:001:0 s a3 00 0000 0001 0004 4 <
520006823 C Ci:2:001:0 0 4 = 00010000
520006856 S Ci:2:001:0 s a3 00 0000 0002 0004 4 <
520006874 C Ci:2:001:0 0 4 = 00010000
520006892 S Ci:2:001:0 s a3 00 0000 0003 0004 4 <
520006909 C Ci:2:001:0 0 4 = 01050100
520007086 S Co:2:001:0 s 23 01 0010 0003 0000 0
```

```
520007146 C Co:2:001:0 0 0
520007216 S Ci:2:001:0 s a3 00 0000 0004 0004 4 <
520007233 C Ci:2:001:0 0 4 = 00010000
520108143 S Ii:2:001:1 -115:2048 4 <
520108274 S Ci:2:001:0 s a3 00 0000 0003 0004 4 <
520108315 C Ci:2:001:0 0 4 = 01050000
520108456 S Co:2:001:0 s 23 03 0004 0003 0000 0
520108492 C Co:2:001:0 0 0
520159161 S Ci:2:001:0 s a3 00 0000 0003 0004 4 <
520159404 C Ci:2:001:0 0 4 = 03051000
520210259 S Co:2:001:0 s 23 01 0014 0003 0000 0
520210289 C Co:2:001:0 0 0
```

The third column consists of four fields delimited by colons. In the first field, `Ci` stands for "control IN", `Co` for "control OUT", `Bi` for "bulk IN", `Bo` for "bulk OUT". Interrupt transfers are coded with a `I`, isochronous transfers with `Z`, with the direction appended as `i` for "IN" and `o` for "OUT".

The second field in that column is the bus number, the third is the device address, and the last is the endpoint number. In this case the host is communicating with endpoint zero of the hub the device was attached to. The hub is on bus 2 and has device address 001.

The rest of the line is transaction specific. An `'s'` at the beginning indicates that this is a setup transaction. The numbers are hexadecimals delimited by whitespace. They show the words in the setup request as described in section A.2.5.

The data and status stages of a control transfer are depicted in a line ending in "`<status> <nbytes> = ...`", where `<status>` is generally `0`, `<nbytes>` is the number of bytes in the data stage, and the ellipsis stands for the individual bytes in groups of four.

Not all transactions are printed, only those that transfer data.

Thus, in this sequence you can see the host polling the hub for the status of the newly attached function until receiving the final status that indicates the function is a high-speed device. The high-speed detection handshake as per section A.1.4 is performed during this time.

A.2.7 Enumeration

Once it is established that the device is properly attached and is acting as a high-speed device, the host starts the enumeration process using a control pipe with endpoint zero, implemented in the file `usb_control.vhd`.

Before enumeration, the device responds to address 0, and only endpoint zero is active. During enumeration the host sets the device address, reads its descriptors, and sets its configuration.

The host first requests the device descriptor, and then resets the device, and only then begins the actual enumeration process. This is done because some functions expect this legacy behavior introduced by the Microsoft Windows operating system. For instance, functions exist that return only 16 of the 18 device descriptor bytes the first time.

The `usbmon` output looks like the following. The second line is continued on the third, indicated here by a backslash and indentation.

```
520210427 S Ci:2:000:0 s 80 06 0100 0000 0040 64 <
520210832 C Ci:2:000:0 0 18 = 12010002 02000040 9afb9afb \
    10000000 0001
520210942 S Co:2:001:0 s 23 03 0004 0003 0000 0
520210962 C Co:2:001:0 0 0
520261274 S Ci:2:001:0 s a3 00 0000 0003 0004 4 <
520261500 C Ci:2:001:0 0 4 = 03051000
520312306 S Co:2:001:0 s 23 01 0014 0003 0000 0
520312335 C Co:2:001:0 0 0
```

In this case the host reads the device descriptor in the first line, and it is returned by the function in the second. The rest is the reset and re-detection traffic with the hub.

To set the device address, the host sends a SETUP transaction of type SET_ADDRESS to endpoint zero.

```
520312361 S Co:2:000:0 s 00 05 0004 0000 0000 0
520312518 C Co:2:000:0 0 0
```

Here, the address is set to 4. From now on the function may only respond to transactions going to address 4.

The host then starts reading the descriptors, starting with the device descriptor.

```
520324323 S Ci:2:004:0 s 80 06 0100 0000 0012 18 <
520324557 C Ci:2:004:0 0 18 = 12010002 02000040 9afb9afb \
    10000000 0001
```

Next, the host requests the configuration descriptor, and, appended to it, all the interface, functional, and endpoint descriptors.

```
520324665 S Ci:2:004:0 s 80 06 0200 0000 0009 9 <
520324793 C Ci:2:004:0 0 9 = 09023e00 02010080 7f
520324832 S Ci:2:004:0 s 80 06 0200 0000 003e 62 <
520324921 C Ci:2:004:0 0 62 = 09023e00 02010080 7f090400 \
    00010202 01000524 00200104 24020005 24060001
```

As you can see, the configuration descriptor is requested twice. The first time only 9 bytes are requested. The third of these (`0x3e = 62`) specifies the total number of bytes including all other descriptors, so only the second time does the host know that it should be requesting 62 bytes.

With the following setup command, the host chooses configuration 1.

```
520325482 S Co:2:004:0 s 00 09 0001 0000 0000 0
520325638 C Co:2:004:0 0 0
```

At this stage, the operating system loads the drivers appropriate to the interfaces of the configuration. In this case, the Linux kernel loads the `cdc_acm` module, which takes control of the device, and sends some device class specific commands pertaining to the state of the physical serial line. Since SWitty does not have a real serial line, these are ignored.

```
520372346 S Co:2:004:0 s 21 22 0000 0000 0000 0
520372545 C Co:2:004:0 0 0
520372592 S Co:2:004:0 s 21 20 0000 0000 0007 7 = 80250000 \
    000008
520372796 C Co:2:004:0 0 7 >
```

The function is now fully enumerated and ready for use.

For completeness, here is the traffic between the host and the hub upon device-detach.

```
530596754 C Ii:2:001:1 0:2048 1 = 08
530596809 S Ii:2:001:1 -115:2048 4 <
530596899 S Ci:2:001:0 s a3 00 0000 0003 0004 4 <
530596934 C Ci:2:001:0 0 4 = 00010100
530597024 S Co:2:001:0 s 23 01 0010 0003 0000 0
530597046 C Co:2:001:0 0 0
530602374 S Ci:2:001:0 s a3 00 0000 0003 0004 4 <
530602400 C Ci:2:001:0 0 4 = 00010000
530628303 S Ci:2:001:0 s a3 00 0000 0003 0004 4 <
530628350 C Ci:2:001:0 0 4 = 00010000
530654326 S Ci:2:001:0 s a3 00 0000 0003 0004 4 <
530654354 C Ci:2:001:0 0 4 = 00010000
530680317 S Ci:2:001:0 s a3 00 0000 0003 0004 4 <
530680345 C Ci:2:001:0 0 4 = 00010000
530706291 S Ci:2:001:0 s a3 00 0000 0003 0004 4 <
530706319 C Ci:2:001:0 0 4 = 00010000
```

A.2.8 Bulk Transfers

Bulk transfers allow the highest data rates of up to 40 MB/s with guaranteed delivery. As opposed to the other transfer types, bulk transfers have the lowest priority, so if many devices with isochronous or interrupt transfers flood the USB, then bulk transfers are slow. Also, if the cable connection is bad, bulk transfers need to be repeated which also results in a lower than optimal transmission rate, but it does insure data integrity.

High-speed bulk endpoints must have a FIFO size of 512 bytes.

The PING flow control mechanism is used to save bandwidth on the USB when sending large data packets to a bulk OUT endpoint. The host sends a PING token packet to the bulk OUT endpoint until the endpoint has room for another 512 bytes and responds with an ACK handshake. If the endpoint does not have room for another 512 bytes, it responds with a NYET handshake. In this way, the bus is only burdened by an up-to-512-byte packet when the endpoint has room for it. Multiple sendings of that packet are reduced to error conditions.[2]

Bulk transfers guarantee data delivery. To achieve this the USB employs data toggle synchronization by using the DATA0 and DATA1 PIDs from Table A.1 in alternating order.

Each side of the USB link keeps a data toggle bit, which is initially set to 0. The bit is toggled whenever the transaction was successful as seen by that side of the link. When no errors occur, the two toggle bits on the two sides of the link are kept in sync.

The sending side will toggle its data toggle bit when it receives a successful ACK handshake for a transaction. The receiving side toggles its bit whenever the data packet is received without error and with the correct toggle bit.

Receiving a data packet with an unexpected toggle bit implies that an error occurred during the handshake, and that the current packet is the same as the last packet. The receiving side must then ignore the packet, yet send an ACK handshake. The sending side must resend the same packet until it receives a valid ACK handshake. That way, the toggle bits on the sending and receiving sides are synchronized.[3]

[2]The PING flow control mechanism is described in section 8.5.1 of the USB 2.0 specification[8].

[3]The toggle mechanism is specified in section 8.6 of the USB 2.0 specification[8].

A.3 USB Device Classes

The USB employs the notion of standardized device classes. The most common classes are the *mass storage device* class used in memory sticks and the *human interface device* class for mice and keyboards. Here, the USB CDC ACM device class is used to emulate a serial device.

A.3.1 Communications Device Class

The USB communications device class was designed for modems and telephone line connectors, and can be used to provide a serial line converter.

With regards to other USB device classes, the communications device class is special in that it adds another complication. It uses multiple interfaces together to act as a single interface, in this context called a *union*.

For the communications device class abstract control model (CDC ACM), two interfaces define the device. In addition, endpoint zero is used for line management. A bulk IN and a bulk OUT endpoint make up the data interface to generically move data across the USB. An interrupt IN endpoint is used for the notification interface for call management in modems, so it is not needed here, and always returns a NAK handshake.

The `cdc_acm` module of the Linux kernel creates the device special file `/dev/ttyACM0`. When writing to it, the USB subsystem directs the data straight to the bulk OUT endpoint of the data interface. Data that the bulk IN endpoint sends is directed towards `/dev/ttyACM0`, and can be captured by reading from that file.

To prevent the `tty` layer to apply special interpretations to the data stream the `stty` command needs to be invoked upon every reboot as follows:

```
$ stty -F /dev/ttyACM0 raw -echo -echoe
```

It is needed to disable special modem interpretation of data characters.

A.4 Conclusion

A simple method to transfer large amounts of data from software to hardware and vice versa was needed. Instead of using hardware that needs custom drivers, the standardized USB CDC ACM device class was used that is supported by many operating systems, including the Linux kernel.

On Linux, a device special file `/dev/ttyACM0` is created upon attachment, which must be initialized after every reload of the `cdc_acm` driver. Receiving

data from the hardware is then as simple as reading from the file, and sending data is as simple as writing to it.

The component is realized as a VHDL module that makes use of a UTMI chip, as is available on many FPGA development boards. As such it is not only simple to use, but also cheap.

In principle, speeds up to 40 MB/s are supported, although great care needs to be taken on the application side in the FPGA as well as the cable quality. In case of error, the USB resends a transaction, which can reduce the transmission rate. Practical speeds are in the 10 MB/s range.

Danksagung

Ich danke allen, die mir bei dieser Diplomarbeit geholfen haben. Die nette Atmosphäre Im Sand hat entscheidend zum Erfolg beigetragen. Ganz besonders danke ich den folgenden Institutsmitgliedern:

Prof. Dr. Andrea Santangelo, der mir das Gebiet der Hochenergieastrophysik vertraut machte und mir diese Diplomarbeit ermöglichte.

Dr. Eckhard Kendziorra, dessen fortwährende Einsatz mich immer gut mit HiWi Stellen versorgte und mich ans Institut holte.

Dr. Christoph Tenzer und **Thomas Schanz**, die mich in die FPGA-Programmierung einführten und durch deren unverzichtbare Expertise die Entwicklung der Elektronik ein leichtes wurde.

Stefan Schwarzburg, dessen Linux-Enthusiasmus und Programmierkompetenz die Weiterentwicklung der `fpipe` zum Spass werden liessen.

Giuseppe Distratis, der mir die Grundlagen des USB Standards auf freundliche Weise in den Kopf hämmerte.

Jürgen Dick, der mir bei der Anfertigung elektronischer Bauteile tatkräftig half.

Daniel Maier für allerlei Diskussionen über dies und jenes und dem SVM Messstand.

Außerdem danke ich meiner Familie, die mich immer unterstützt hat. Auch danke ich allen Mitgliedern des Instituts, die mit mir Fußball gespielt haben so lange es ging und die Kaffeepause mit Diskussionen, Experimenten und Wettbewerben ausschmückten und mir ermöglichten die Lichtgeschwindigkeit zu messen.

Vielen, vielen Dank!

Plagiaterklärung

Hiermit erkläre ich, Henry Sebastian Grasshorn Gebhardt, dass ich die Diplomarbeit mit dem Titel *Development of Data Acquisition and Detector Controller Electronics for the Low Energy X-Ray Detector of the Simbol-X Space Mission* selbständig verfasst und dabei keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Tübingen, den 1. Dezember 2009