EPSON
EXCEED YOUR VISION

S1V30120
Getting Started Guide for System Developers

SEIKO EPSON CORPORATION

Rev.1.00

NOTICE

No part of this material may be reproduced or duplicated in any form or by any means without the written permission of Seiko Epson. Seiko Epson reserves the right to make changes to this material without notice. Seiko Epson does not assume any liability of any kind arising out of any inaccuracies contained in this material or due to its application or use in any product or circuit and, further, there is no representation that this material is applicable to products requiring high level reliability, such as, medical products. Moreover, no license to any intellectual property rights is granted by implication or otherwise, and there is no representation or warranty that anything made in accordance with this material will be free from any patent or copyright infringement of a third party. This material or portions thereof may contain technology or the subject relating to strategic products under the control of the Foreign Exchange and Foreign Trade Law of Japan and may require an export license from the Ministry of Economy, Trade and Industry or other approval from another government agency.

All other product names mentioned herein are trademarks and/or registered trademarks of their respective companies.

©SEIKO EPSON CORPORATION 2007, All rights reserved.

Table of Contents

1. Introduction & Scope **1**

2. Overview of the S1V30120 .... **2**

2.1 Key Functionality 3

2.2 System Integration 3

3. System Level Considerations **4**

3.1 Introduction 4

3.2 System Memory requirements for S1V30120 Integration 4

3.2.1 System Memory Requirements Calculation 6

3.3 Host CPU – S1V30120 Serial Link Interface: System Considerations 7

3.3.1 Connectivity Considerations 7

3.3.2 Link Speed requirements: Real time Constraints 9

3.3.3 Initialization Time considerations 11

3.4 Analogue Output considerations 12

3.4.1 Analogue Power & external circuitry 12

3.4.2 Recommendations for volume levels 12

4. Developing content for your application **13**

4.1 Introduction 13

4.2 TTS Content 13

4.2.1 Overview of TTS functionality 13

4.2.2 TTS Content Development: basic flow using EPSON Evaluation Kit 15

4.3 ADPCM 17

4.3.1 Overview of ADPCM Functionality 17

4.3.2 ADPCM Content Development 17

5. Integrating and accessing S1V30120 data in the system **19**

5.1 Introduction 19

5.2 Packaging S1V30120 data files: EPSON S1V30120 Packaging utility 19

5.2.1 S1V30120 Data Packaging tool usage 20

5.2.2 S1V30120 Data Packaging tool Output Files 21

5.2.3 Accessing S1V30120 data: an example API ... 25

6. Developing driver code for S1V30120: a simple TTS streaming driver... 31

6.1 Introduction ... 31

6.2 Example S1V30120 driver architecture... 32

6.3 Developing Message Rx/Tx Handlers for the Host CPU: an example SPI Channel driver ... 34

6.4 Streaming Data from Host CPU to S1V30120: an example S1V30120 event handler for TTS streaming ... 36

6.5 Developing for low performance Host CPUs : Half Duplex Operation ... 37

1. Introduction & Scope

This document is an introduction to the S1V30120 Voice Guidance IC, aiming to provide an initial understanding regarding its operation, the requirements which imposes on your system and content/firmware development for system level integration. The document is structured as follows:

- Section 2 presents an overview of the device, its main features, functionality, and key advantages for system integration
- Section 3 offers an overview of the key requirements for S1V30120 system integration. Specifically, the following aspects are examined:
  - System Storage requirements for S1V30120 data
  - Host CPU – S1V30120 communication: connectivity, bit-rates and initialization time considerations
  - S1V30120 Analogue Output
- Section 4 provides insight into audio speech content development for S1V30120. It focuses on TTS and ADPCM content creation
- Section 5 focuses on how to structure and access content/initialization data within the Host CPU memory and provides an example solution for this task
- Finally, section 6 examines issues related with Host CPU firmware development for S1V30120 drivers, and provides illustrative examples for a very simple TTS streaming driver.

2. Overview of the S1V30120

The S1V30120 is a Voice Guidance IC designed to add speech/audio output capabilities to systems for minimal cost and with minimal integration complexity overhead. The following table depicts the main features of the device:

Table 1 S1V30120 – summary of functionality and features

<table>
<tr><th colspan="4">Functionality</th></tr>
<tr><td rowspan="7">TTS Synthesiser</td><td>Synthesiser Details</td><td colspan="2">Fonix Dectalk TTS (v5.0.e1) fully integrated (royalty free) in the device.</td></tr>
<tr><td rowspan="2">Language support</td><td colspan="2">US English</td></tr>
<tr><td colspan="2">Spanish (Castilian & Latin-American variants)</td></tr>
<tr><td>Synthesis Input</td><td colspan="2">Standard ASCII Text, streamed via a serial interface by a host CPU.</td></tr>
<tr><td>Synthesis Output</td><td colspan="2">11.025kHz Mono Analogue Output</td></tr>
<tr><td rowspan="2">Customization Options</td><td colspan="2">Modification of speech output parameters (speed, gender, pitch, prosodic parameters), by means of
- configuration messages issued by the host CPU
- mark-up symbols embedded in the input text</td></tr>
<tr><td colspan="2">User Dictionary: for English/Spanish words which are not present in the internal Fonix dictionary within the S1V30120, it is possible to download a customized pronunciation dictionary from the host CPU into the S1V30120.</td></tr>
<tr><td rowspan="3">ADPCM Decoder</td><td>Decoder characteristics</td><td colspan="2">ITU G.726 compliant ADPCM decoder</td></tr>
<tr><td>Supported input bit rates</td><td colspan="2">80kbps,64kbps, 48kbps, 40kbps, 32kbps and 24kbps</td></tr>
<tr><td>Supported audio sampling frequencies</td><td colspan="2">8kHz,11.025kHz,16kHz</td></tr>
<tr><th colspan="4">Interfaces & System Integration</th></tr>
<tr><td rowspan="2">Host CPU Interface</td><td>Physical characteristics</td><td colspan="2">single 3-wire, SPI slave interface for both control & data</td></tr>
<tr><td>Logical</td><td colspan="2">Unified message protocol for control and data streams from host CPU to S1V30120, and status/feedback from S1V30120 to host CPU</td></tr>
<tr><td>Analogue Output Interface</td><td colspan="3">16 bit input, single channel audio DAC with D-class output amplifier (11mW output power)</td></tr>
<tr><th colspan="4">Hardware Features</th></tr>
<tr><td colspan="2" rowspan="2">Supply Voltages</td><td>HVDD,AVDD (Core, Analogue)</td><td>3.3V</td></tr>
<tr><td>LVDD,PLLVDD (Core,PLL)</td><td>1.8V</td></tr>
<tr><td colspan="2">Clock Input Frequency</td><td colspan="2">32.768kHz</td></tr>
<tr><td colspan="2">Typical Active Power Consumption</td><td colspan="2">40 mW (See Note 1)</td></tr>
<tr><td colspan="2">Typical Static Power Consumption</td><td colspan="2">30 μW (See Note 2)</td></tr>
<tr><td colspan="2">Package</td><td colspan="2">64-pin TQFP (0.5mm pitch)</td></tr>
</table>

Note 1: Ambient Temperature=25°C, DAC output unloaded, S1V30120 performing TTS synthesis. Note that this is a guideline value: power consumption can vary depending on TTS speech output speed and other synthesis parameters.

Note 2: Ambient Temperature = 25°C

2.1 Key Functionality

The S1V30120 provides two basic mechanisms for prompting the user of the system user interface output:

- Unconstrained TTS [Text-To-Speech] Synthesiser for English and Spanish languages. The device can synthesise highly intelligible speech output from an input ASCII-text stream, and provides several options for the customization of the speech/voice characteristics: voice gender/timbre, speech speed, and several other characteristics can be tuned for the specific needs of your system. Moreover, for English/Spanish words that are not present in the internal S1V30120 dictionary, it is possible to generate (using utilities provided by EPSON and Fonix Corporation) a customized pronunciation dictionary and download it to the S1V30120. Consult section 4.2 for an in-depth introduction to system development using the TTS synthesiser in the S1V30120.
- ADPCM Decoder: although TTS enables your system to provide verbal information to users, sometimes it is necessary to generate non-verbal prompts such as beeps, alarm bells or start-up tune sounds. These non-verbal cues enhance the system user experience. The S1V30120 incorporates an ADPCM decoder capable of playing back such cues, from an input ADPCM stream. The decoder is ITU G.726 compliant, and supports several bit-rates and audio sampling frequencies. For more information on the ADPCM Decoder, refer to section 4.3.

2.2 System Integration

One of the key advantages of the S1V30120 is its ease of integration into your system:

- S1V30120 is integrated as a slave peripheral of your system's CPU –the Host CPU-. As a purely reactive decoder/synthesiser, the S1V30120 effectively behaves as a black box, which generates analogue output when presented with an input stream. This means that the system developer need only concern themselves with the development of firmware drivers for the S1V30120 to be deployed at the Host CPU. Knowledge of the internal architecture of S1V30120 or development of code for it is not required.
- Storage of ADPCM data or TTS input text files is managed by the system CPU, not by the S1V30120. This reduces chip count, and simplifies the upgrading and maintenance of a speech user interface (Consult section 3.2 for the system memory requirements imposed by integrating a S1V30120 into your system).
- Both commands and data are streamed into the S1V30120 using a unified, single message protocol specification, and single SPI serial interface. This eases the task of the system developer by providing a uniform logical and physical interface for all data flow between system CPU and S1V30120, which simplifies the development of firmware at the system CPU to drive the S1V30120. For details of the S1V30120 Message Protocol, consult the document "S1V30120 Message Protocol Specification", included in this Evaluation kit release.
- The S1V30120 integrates a 16 bit mono Digital-to-Analogue Converter [DAC]. The DAC is fully controllable (volume, sampling rate) by commands using the SPI Host-CPU interface.

3. System Level Considerations

3.1 Introduction

The first step in the evaluation of the S1V30120 solution is to assess whether your system meets minimum requirements for S1V30120 integration. Also how including the device will affect the resources and BOM of your system. This section offers detailed information on the following key issues regarding integration:

- System Memory requirements for S1V30120 data
- Host CPU – S1V30120 communication: connectivity, data rates and initialization time requirements
- Analogue output: external component requirements

3.2 System Memory requirements for S1V30120 Integration

As explained in section 2.2, the S1V30120 does not include non-volatile memory: all data (text files for TTS, ADPCM files, TTS user dictionaries) must be streamed to the S1V30120 via the Host CPU serial interface.. This implies that system designers have to allow for storage of this data in the system non-volatile memory (Flash or EEPROM in typical embedded systems). The global requirements in terms of storage can be subdivided into the following components:

- **TTS text files:** The S1V30120 accepts plain ASCII text files as input for its TTS synthesiser, which has to be streamed into the device by the Host CPU. A guideline for memory usage (equally applicable to both English and Spanish) could be as follows: rough memory storage required for TTS text files is ~6Bytes/word or 1.2 KB per minute of speech, at average English/Spanish speaker rates.
- **ADPCM stream files:** ADPCM storage requirements vary depending on the bitrates of the stream. For example, a 16 kHz, 80 kbps will give the highest possible quality, but at the cost of 8KB storage per second of audio output, while 24kbps ADPCM (3KB/second) trades off audio quality for lower storage requirements.
- **User Dictionaries:** If the end application requires the use of non-standard words in English or Spanish, EPSON and Fonix Corporation provide tools (see “Fonix Dictionary Build Tool User Guide” document) to generate a customized User Dictionary from a text file containing the customized pronunciations. A User Dictionary is essentially a binary representation of the text file that can be downloaded into the S1V30120. For average Spanish/English utterances, the storage requirements for each pronunciation entry in a User Dictionary is roughly 15 Bytes/Word, with a maximum allowable size of 2KB per dictionary (the maximum amount of customized words is ~150).
- **Initialization Data**: The S1V30120 requires specific initialization data to be downloaded from the Host CPU after reset. This data is provided to customers in the form of a binary file (S1V30120_INIT_DATA) which can be found within the installation folder of this evaluation kit)

that must be streamed via the Host CPU SPI interface (consult section S1V30120 Message Protocol Specification, section 4). 32 KB are required for storage of the S1V30120_INIT_DATA file.

Table 2 summarises the memory requirements associated with the integration of the S1V30120 into a system:

Table 2 Host CPU storage requirements for S1V30120 Data

<table>
<tr><th>Data Type</th><th colspan="2">Storage Requirements</th></tr>
<tr><td>TTS text input files</td><td colspan="2">~6 Bytes/Word
~1.2 Kbytes/minute of audio output</td></tr>
<tr><td rowspan="5">ADPCM input streams</td><td>Bitrate (quality)</td><td>Kbytes/second of audio output</td></tr>
<tr><td>64kbps</td><td>8KB</td></tr>
<tr><td>48kbps</td><td>6KB</td></tr>
<tr><td>32kbps</td><td>4KB</td></tr>
<tr><td>24kbps</td><td>3KB</td></tr>
<tr><td>TTS User Dictionary</td><td colspan="2">15Bytes/Word
Total Size < 2KB</td></tr>
<tr><td>Initialization Data
(S1V30120_INIT_DATA)</td><td colspan="2">32KB</td></tr>
</table>

3.2.1 System Memory Requirements Calculation

As an example of system memory requirements, the storage required for the integration of the S1V30120 into a hypothetical application will be considered. These are the audio files required by the application:

- TTS Input text files: Complete appliance instruction manual and speech driven menu system – approx 15min of speech (~3000 words).
- TTS User Dictionary: specific pronunciations for 35 common SMS abbreviations in English and Spanish.
- ADPCM audio files: 2 medium quality ‘chimes-like’ tunes for power-up and shut-down (3-seconds each), 2 short ‘beep’ sounds in lowest quality, for alarm and malfunction signals.

Using the information in table 2 the calculation for total system memory requirements is as follows:

Table 3 Example of system storage requirements for a hypothetical application

Data Type	Storage Requirements		Application requirements	Data Size
TTS text input files	~6 Bytes/Word ~1.2 Kbytes/minute of audio output		**15 minutes of speech (~3000 words)**	18KB
ADPCM input streams	Bitrate (quality)	Kbytes/second of audio output		
	64kbps	8KB		
	48kbps	6KB	**2 x 3sec files (chimes)**	36KB
	32kbps	4KB		
	24kbps	3KB	**2x1sec files (beeps)**	6KB
TTS User Dictionary	15Bytes/Word Total Size < 2KB		**2 x 35 words (SMS abbr. in English & Spanish)**	1KB
Initialization Data (S1V30120_INIT_DATA)	32KB		**Initialization Data is Mandatory for S1V30120 operation**	32KB
			TOTAL	**93KB**

3.3 Host CPU – S1V30120 Serial Link Interface: System Considerations

The S1V30120 is a purely reactive slave device, with all the control and data messages issued from a Host CPU via a SPI serial link. This section outlines the requirements that the S1V30120 integration imposes on the serial link of the Host CPU.

3.3.1 Connectivity Considerations

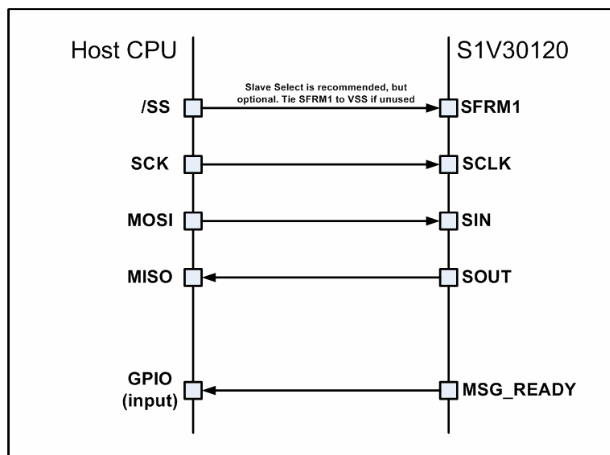

Fig. 1 Connectivity between a Host CPU and the S1V30120 device

In order to establish proper communication between the S1V30120 and Host CPU, the following connectivity is required (as depicted in figure 1):

- SPI master interface at the Host CPU side. As the S1V30120 is an SPI slave, the Host CPU is required to drive the clock line of the SPI link. The following recommendations apply:
- Regarding format, POL=1 (clock idle state is low) PH=1 (first edge of the clock is used to sample the data) is required for use with the S1V30120. The following figure shows waveforms for an SPI link configured with POL=PH=1

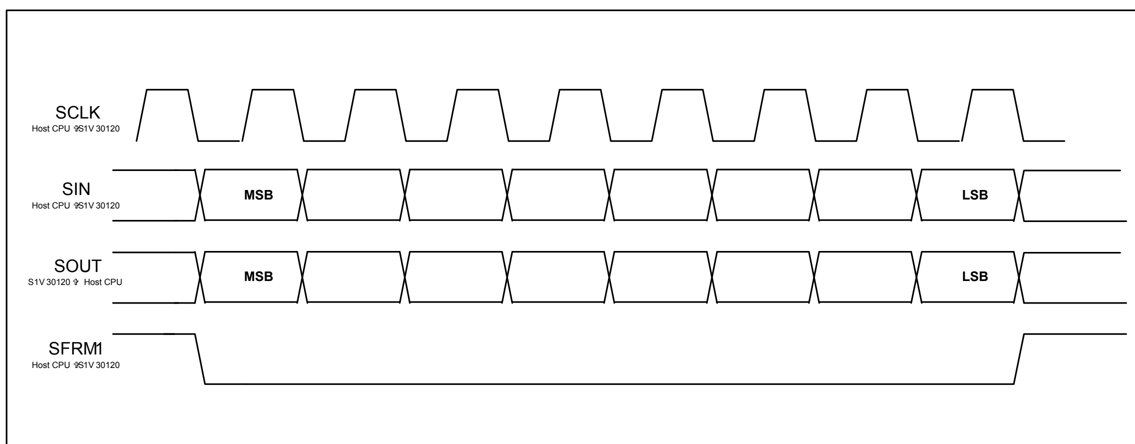

Fig. 2 waveforms for the Host CPU-S1V30120 SPI link configured with POL=PH=1

- Use of a Slave Select signal (SFRM1 pin in S1V30120, consult S1V30120 Hardware Specification, section 3) is recommended but not necessary for correct operation of the link. In other words, it is possible to operate the S1V30120 SPI link as a simple 3-wire serial interface: just make sure that SFRM1 line in S1V30120 is permanently tied to VSS.
- A GPIO input at the Host CPU side, connected to the S1V30120 MSG_READY line: the logical message protocol built on top of the physical SPI interface relies on a simple Host request – S1V30120 acknowledgement/response paradigm. As SPI transfers can only be initiated by the SPI Master (the Host CPU starting the SCLK clock), ,a mechanism for informing the Host CPU that a response is ready to be sent by the S1V30120 is required. The MSG_READY line implements that mechanism: a positive edge (VSS→HVDD transition) at the MSG_READY line in S1V30120 indicates that a response/indication is ready to be sent to the Host CPU. Therefore, it is recommended that the GPIO line at the Host side can generate edge-triggered interrupts, for timely Host CPU response time.

Table 4 summarizes the connectivity requirements on the host side for S1V30120 integration:

Table 4 Connectivity requirements between a Host CPU and the S1V30120

Host CPU Requirement	Functionality	System Considerations
1x SPI Master Interface	Physical link for S1V30120 control message protocol	Ensure that the Master SPI block operates in PH=POL=1 Mode (figure 2)
		If a 3-wire style link is used (no Slave Select), ensure that SFRM1 pin in S1V30120 is tied to VSS
1 x GPIO Input	Provides a mechanism for signalling to the host CPU when the S1V30120 is ready to send a message	Connect the line to MSG_READY pin in S1V30120
		MSG_READY signals a response with a rising edge: make sure that the GPIO at the host side can generate edge-triggered interrupts

3.3.2 Link Speed requirements: Real time Constraints

An important requirement on the Host CPU is the minimum bit-rate that is required to meet real time constraints when streaming data from it to the S1V30120. TTS is not a concern in terms of real time: as the input to the device when synthesising speech is just text, the bandwidth requirements are really low. When streaming audio encoded in ADPCM format, the situation is quite different: the worst possible case occurs when the highest encoding quality takes place. The stream itself has a bit-rate of 80kbps, but the overheads of message request/response passing to and from the S1V30120 must be taken into account. Fig. 3 shows a typical ADPCM streaming situation (details of message sequences for ADPCM streaming can be found in the S1V30120 Message Protocol Specification document, section 5.8):

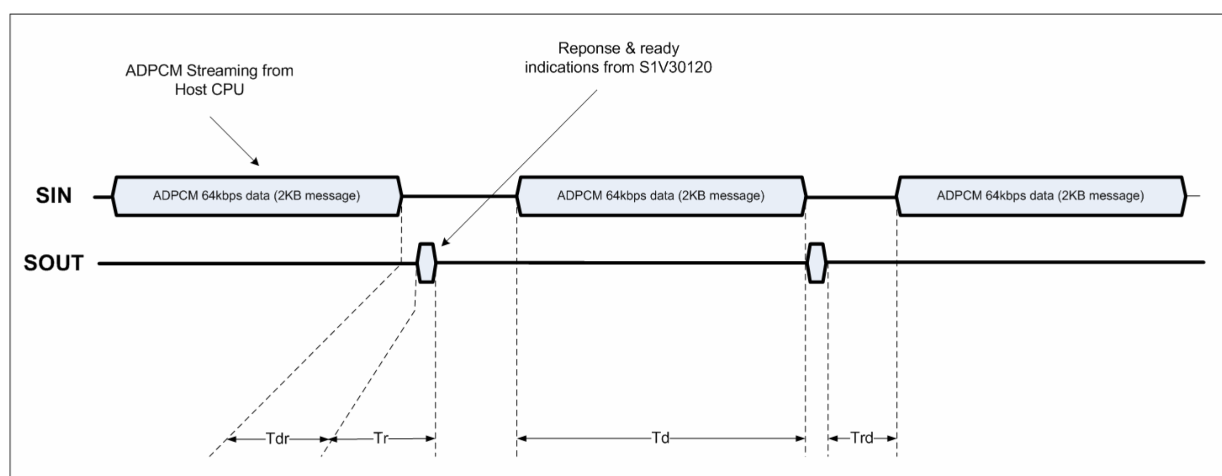

Fig. 3 relevant times for real-time constraints in Host CPU-S1V30120 link minimum SPI clock rate

The figure shows the relevant times that determine the actual clock-rate in the SPI link:

- Td: Time required to stream an ADPCM payload of maximum (2Kbytes) length
- Tr: Time required to send the sequence of responses to the ADPCM data from the S1V30120
- Trd: Time between the Host CPU receiving response and indication that new data can be sent, and the start of the next ADPCM payload.
- Tdr: Typical time between an ADPCM payload being sent and S1V30120 sending a response.

It's clear that the two parameters within the control of the Host CPU is the SPI link clock rate and the delay between receiving a S1V30120 response and sending a new block of ADPCM data (i.e. Trd). To give a clear idea of the requirements of clock rate and delays at the Host CPU side, figures 4 and 5 provide limits for correct ADPCM streaming operation for both parameters. Exceeding these limits will result in broken audio playback at the S1V30120 side.

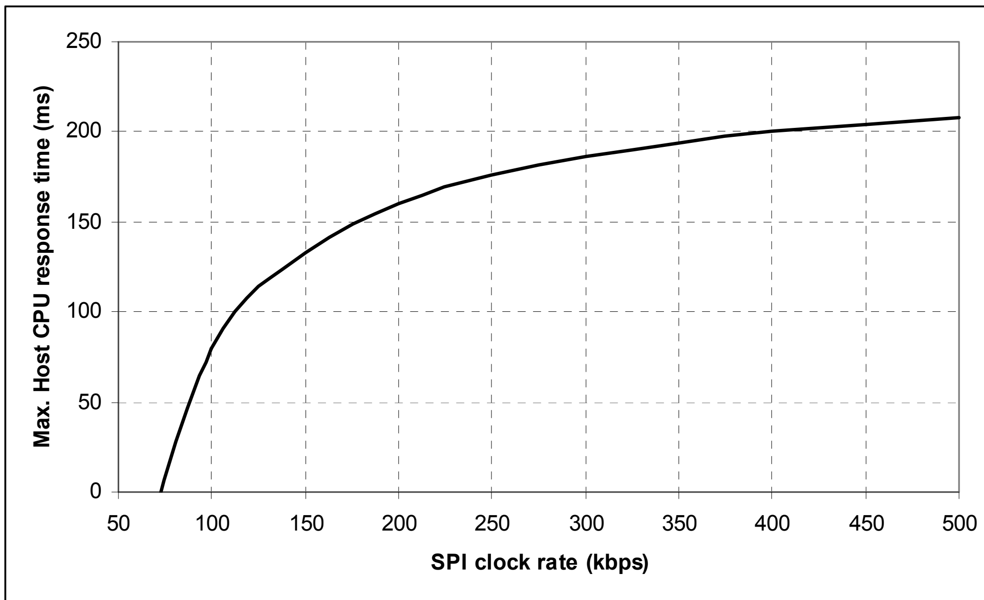

Fig. 4 Max. Delay between S1V30120 response to an ADPCM data message and Host CPU sending a new block of data. Bigger average delays will result in broken audio output. The times are given for minimum communication overhead (ADPCM data block size maximum: 2048 bytes)

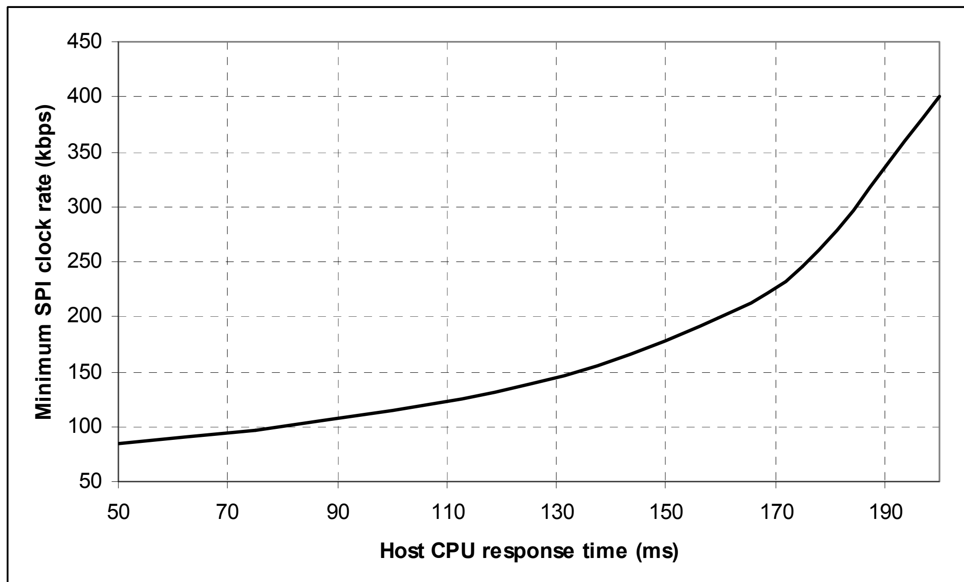

Fig. 5 Minimum SPI clock rate required for 64kpbs ADPCM decoding for a given Host CPU response time to S1V30120 indicating that new data can be sent. Lower clock rates will result in broken audio output. The times are given for minimum communication overhead (ADPCM data block size maximum: 2048 bytes)

3.3.3 Initialization Time considerations

The S1V30120 device must be initialized prior to being operational. The following steps take place during initialization, once the device is powered up and comes out of reset:

Stabilization of S1V30120 internal PLL output

- The initialization data, S1V30120_INIT_DATA is streamed via the Host CPU serial link (see S1V30120 Message Protocol, section 4.3 - figure 3)
- The S1V30120 boots up, and is fully operational

The time required for initialization is again mainly a function of the response time of the Host CPU to message responses from the S1V30120 and the clock rate of the SPI serial link. Fig. 6 shows the initialization times (i.e. time from reset de-assertion to the device fully operational) for different Host CPU response times and SPI clock rates:

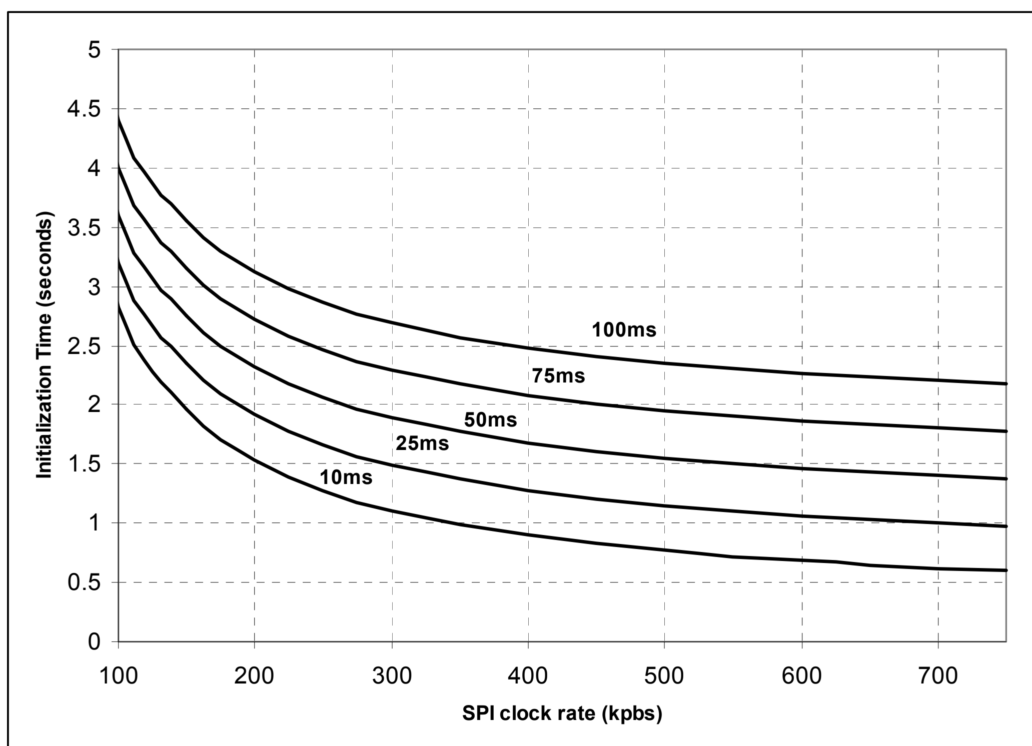

Fig. 6 S1V30120 initialization time vs. SPI link clock rate for different Host CPU response times.

3.4 Analogue Output considerations

3.4.1 Analogue Power & external circuitry

For the output of speech and audio signals, the S1V30120 incorporates a 16 bit mono DAC. However, the internal DAC can only drive up to 11mW load within acceptable distortion levels. This implies that if your system requires output power levels higher than typical headphone levels (i.e. small, cheap speakers in appliances typically require hundreds of mW), an external audio amplification circuit has to be included.

EPSON recommends a standard output amplifier circuit described in the document "S1V30120 Application Note – Application Circuit Example", section 3, included in this evaluation kit release. The circuit uses an inexpensive 386 amplifier (available from several manufacturers: National Semiconductor, New Japan Radio, etc), and a few external passive components, so the impact on your BOM will be minimal.

3.4.2 Recommendations for volume levels

Output level is strongly dependant on the actual speaker used for the final product where the S1V30120 is integrated. However, it is worth noting the following guidelines:

- When using the audio circuit recommended in "S1V30120 Application Note – Application Circuit Example", a volume setting of 0dB will deliver a near full-scale output waveform in a low power, passive speaker, for TTS speaking.
- For ADPCM streaming, if the initial PCM file from where the ADPCM was generated had levels close to full scale, the output waveform will also be near full scale with a S1V30120 volume setting of 0dB.

4. Developing content for your application

4.1 Introduction

After considering the requirements for S1V30120 integration and the impact on your system, the next step is to develop audio and speech content for your application, and customize this content to meet your quality/memory footprint requirements. EPSON provides all the tools required for the creation of content in this evaluation kit release:

- ADPCM Encoder tool for the generation of audio content
- Evaluation Board, and PC-GUI software to customize and test the quality of ADPCM and TTS content
- User Dictionary Utility for the creation of customized TTS user dictionaries

The following sections describe in detail how to customize TTS and ADPCM contents to meet your application requirements using these tools.

4.2 TTS Content

4.2.1 Overview of TTS functionality

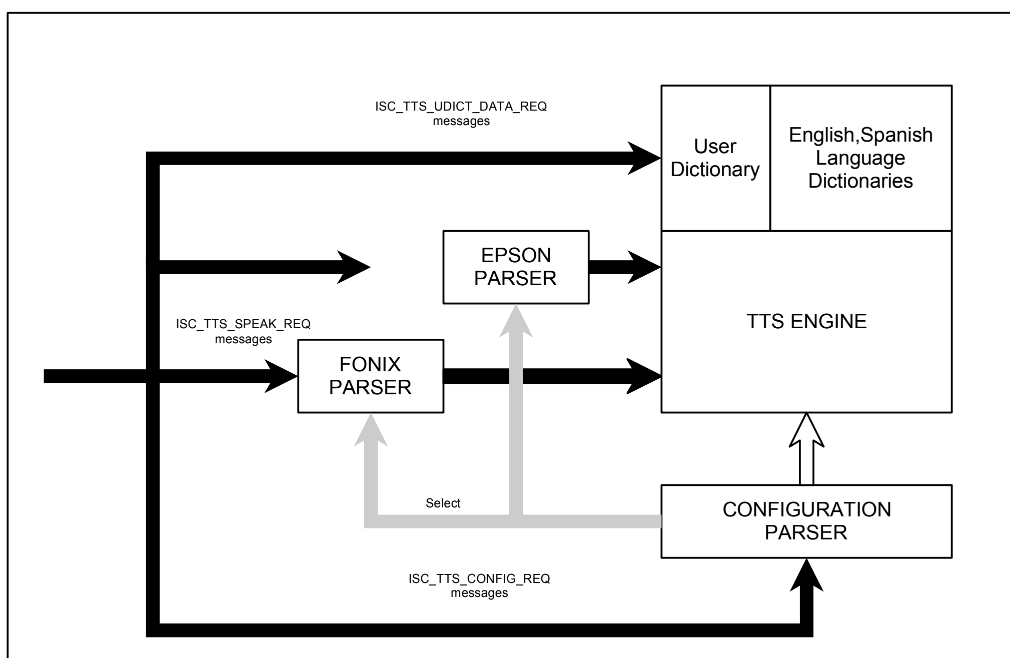

Fig. 7 Conceptual representation of the S1V30120 TTS engine

At a first glance, the S1V30120 is basically a text-input, speech-output engine. However, in order to customize the TTS engine, a deeper insight into the internals of the TTS functionality is required. Fig. 7 shows a conceptual representation of how the S1V30120 TTS engine works: the TTS core functionality is accessed via 3 different components,

which control the different parameters for speech output production, and includes a configurable user dictionary area, that can be accessed from the Host CPU. All of these components allow speech customization, in progressive levels of granularity:

- **Configuration parser**: the simplest, coarser way of controlling speech output parameters is by using ISC_TTS_CONFIG_REQ messages (see section 5.9.3.1 of S1V30120 Message Protocol Specification document, included in this evaluation kit release). These messages control very basic features of the TTS synthesizer:

  Selection of Preset Voices

  - Type of text parser to be used (EPSON or Fonix)
  - Language (Spanish or English)
  - Speaking Rate (words per minute)

  It's worth noting that once a configuration message is sent, the settings will be applied to any subsequent text streamed into the S1V30120, unless they are overridden by mark-up parameters embedded in the text. In other words, while the configuration messages have session scope, configuration embedded in a text file applies to that file only (file-scope).

- **EPSON parser:** this parser allows finer control of the characteristics of the speech output, allowing dynamic changes in emphasis, pitch, voice selection, and speaking rate to take place within a text. This is achieved by embedding mark-up control symbols in the text itself (consult section 5.9.1.5 of S1V30120 Message Protocol Specification). Note that this functionality is *incompatible* with the fine-granularity control offered by the Fonix parser: the ISC_TTS_CONFIG_REQ message selects either EPSON or Fonix parser, and the functionality of the two parsers *cannot* be mixed.

- **Fonix parser:** for the finest granularity in dynamic customization, the Fonix parser offers direct access to the internal parameters of the TTS engine. Dynamic customization using the Fonix parser is relatively involved, but it offers the highest level of control. Specific details for the mark-up symbols and the parameters they control can be found in the "FONIX DECtalk (R) 5.01-E1 User Guide" document

- **User Dictionaries:** On normal operation, the TTS engine uses an internal dictionary and a pronunciation rule-set for the language selected by the most recent configuration message. However, maybe the application requires some specific abbreviations or words be synthesized to a customer defined pronounciation...For this case, it is possible to install a user dictionary with the pronunciation for these abbreviations/custom words. Details about the creation of user dictionaries can be found in the "Fonix Dictionary Build Tool User Guide" document within this evaluation kit release.

4.2.2 TTS Content Development: basic flow using EPSON Evaluation Kit

Once that the internals of the TTS engine in the S1V30120 are understood, it is straightforward to develop speech content for your application. Using the evaluation kit tools (S1V30120 Evaluation GUI software + Evaluation Board), the development flow would be as follows:

Install the S1V30120 Evaluation GUI Software and make sure that it works correctly in conjunction with the S1V30120 Evaluation Board. Instructions are provided in the S1V30120 Evaluation Kit User Guide document provided with this evaluation kit release.

1. Initially, use some generic text and select the general settings that suit best your application, using the GUI controls that set language, voice selection and speaking rate. Note that these settings can be specified in an ISC_TTS_CONFIG_REQ message, and will apply for the whole TTS section. Therefore, you should take note of these settings, as they will be used in messages sent by your host CPU to the S1V30120 in your final product.
2. Once your basic session settings are selected, copy/paste your application text content into the text box of the GUI and check its quality. If the results are satisfactory, proceed to step 5
3. If specific parts of the text need to be sped-up, emphasized, or pronounced with a different voice, select the EPSON parser in the GUI controls and embed the necessary mark-up symbols –as described in section 5.9.1.5 of S1V30120 Message Protocol Specification – to achieve the required speech prosody. Experiment with the mark up until the required prosody is achieved.
4. If the results achieved by the EPSON parser are not satisfactory, further fine tuning of speech prosody and phoneme pronunciation control is required. Switch to the Fonix parser using GUI controls, and experiment with the mark-up control symbols as described in the “FONIX DECtalk (R) 5.01-E1 User Guide” document.
5. If application specific abbreviation/words that are not commonly used in the selected language are required, generate a User Dictionary by using the S1V30120 User Dictionary Tool (consult “Fonix Dictionary Build Tool User Guide” for information on this procedure)
6. Store the resulting text files and user dictionaries and keep note of all the TTS GUI configuration settings. These contents will be streamed from the Host CPU to the S1V30120 in your final product, using pre-specified sequences of request/response messages defined in the S1V30120 Message Protocol documentation:
   - Configuration settings: ISC_TTS_CONFIG_REQ/RESP (section 5.9.3.1-2, fig. 22)
   - Storing of User Dictionary: ISC_TTS_UDICT_DATA_REQ/RESP (sections 5.9.3.17-18 , fig. 30)

- Streaming of TTS text: ISC_TTS_SPEAK_REQ/RESP, ISC_TTS_READY_IND, ISC_TTS_FINISHED_IND (sections 5.9.3.3-5.9.3.6, fig. 23 & 31)

4.3 ADPCM

4.3.1 Overview of ADPCM Functionality

The S1V30120 is capable of encoding audio effects (such as beeps, tunes and alarm sounds) to further enhance the speech interface in your application. The ADPCM implementation is ITU G.726 compliant, and supports the customary 24, 32 and 40 kbps bit-rates @ 8 kHz output sampling rate, plus a specific high quality 80kbps sampling rate that will generate audio output @ 16 kHz sampling rate.

4.3.2 ADPCM Content Development

For ADPCM content development, EPSON provide an ADPCM encoder tool, included in this evaluation kit release, which will generate files ready to be split into message payloads and streamed via the SPI link (see the S1V30120 Message Protocol Specification, section 5.8 for details on ADPCM streaming). The following considerations must be taken into account:

- Input file: the EPSON ADPCM encoder tool accepts input files with the following characteristics:
  - File Format: Raw PCM
  - Audio Sample Format: Mono, 16-bit, Intel (little endian) byte ordering
  - Audio Sampling Rate: 8 kHz or 16 kHz, depending on the sampling rate required for S1V30120 output.
- ADPCM encoder tool usage: the EPSON ADPCM encoding tool can be found in the `/ADPCMEncoder` folder within this evaluation kit release. Usage is as follows:

  common_alg_file_client.exe –d encode –w SPCODEC –t adpcm –b <*bitrate*> -s <*sampling_freq*> -c 1 –i <*input_filename*> -o <*output_filename*>

Where the parameters to be given are these:

- **<input_filename>:** the name of the raw PCM input file
- **<output_filename>**: the name of the ADPCM output file
- **<sampling_freq>,<bitrate>**: input file sampling rate in Hz (8000,16000), and encoding bit rate in kbps -24,32,40,48, 64,or 80-, respectively. The following table shows the allowed combinations of these factors:

Table 5 Allowable combinations of bit-rates and input/ output sampling frequencies for ADPCM encoding

Input file sampling Rate (Hz)	Allowed bit rates (kbps)	Decoded output sampling rate (Hz)
8000	24,32,40	8000
16000	48,64,80	16000

For selection of bit-rate parameter, a trade-off between speech quality and memory usage (see section 3.2) has to be made. It is recommended that you try different combinations of bit-rate/sampling frequency in the S1V30120 Eval Kit GUI + Evaluation Board (see S1V30120 Evaluation Kit User Guide document) to find the combination that best fits the application quality requirements within acceptable memory footprint.

5. Integrating and accessing S1V30120 data in the system

5.1 Introduction

Once the audio and speech content for the application has been prepared, the next step is to create a system prototype integrating both your Host CPU subsystem (CPU + memory) and the S1V30120. The first issue to be addressed is how to organize and store the content and initialization data in a way that facilitates straightforward streaming of it to the S1V30120. In this section, a tool (the S1V30120 Data Packaging Utility, provided in this eval kit release) and code for a clear API interface that will help you do so is presented. Please note that this is just an *example* of how the data might be structured therefore system developers are not required to use this framework:

- In a high-end system, the application is likely to run on top of an RTOS/OS. In such case, a file system will be available to system developers: there is no need to provide a specific API for S1V30120 data, as it can be conveniently accessed via file system facilities.
- Low end platforms (e.g. MIPS-limited 8-bit CPUs), might require to store S1V30120 data already subdivided into fully packaged messages (with Start-of-Message commandls and message headers - see S1V30120 Message Protocol Specification document), so that they can be streamed with minimal MIPS usage. In this case, a custom data access API must be developed. Nevertheless, the material in this section can still be useful as a guideline for the development of your custom API.

5.2 Packaging S1V30120 data files: EPSON S1V30120 Packaging utility

Once the speech and audio interface has been designed, the system developer has several files that must be stored in the system's non-volatile memory, accessed by the Host CPU, and streamed to the S1V30120 as required. In order to facilitate this task, EPSON provides a utility that packages-up all the files into a single ANSI-C constant array (throughout this section it is assumed that Host CPU software development is carried out using the ANSI-C language). Furthermore, the Data Packaging utility generates C files with data structures that greatly simplify data access and streaming via messages over the Host CPU serial link.

Fig. 8 shows an overview of the development flow:

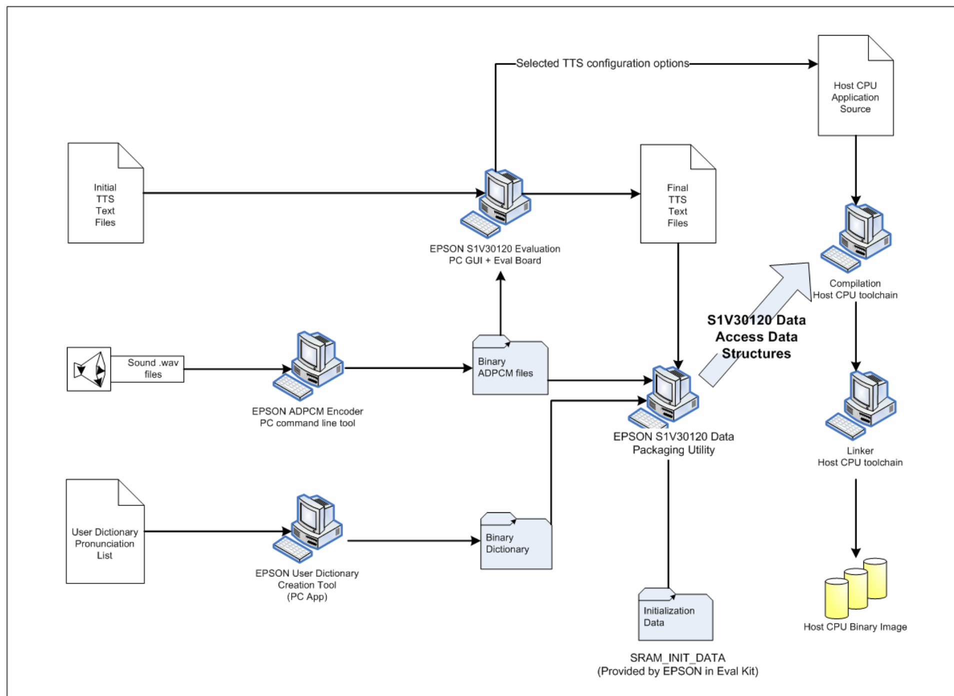

Fig. 8 Development flow for S1V30120 System Integration

5.2.1 S1V30120 Data Packaging tool usage

The Data Packaging tool is a simple command line application that is invoked from a DOS window (the executable and libraries required for it to run correctly can be found in the folder /S1V30120_Data_Packaging within this evaluation kit release. Note that the tool consists of the executable itself, plus a Python runtime DLL and compressed library file: the tool does not work without all these files in the same folder). Usage format is as follows: open a Windows command line console, go into the /S1V30120_Data_Packaging folder in the S1V30120 Evaluation Kit installation folder, and type:

```
S1V30120_data_pack.exe <input file list> <payload size>
```

Where the following parameters are given:

`<input>`: input is the name of a text file containing a list of data files to be included in the final output data. This will include its path relative to the Data Packaging Utility. Each input file must be provided on a separate line within the list file. Furthermore, *TTS text files must have the extension .txt*. For the example explained in section 3.2.1 (memory usage), the input file would be similar to the following (the path for each file is relative to the data packaging tool path):

```
../data/S1V30120_INIT_DATA
../data/beep.adpcm
../data/TTS_user_dict.bin
../data/hello.txt
../data/manual1.txt
../data/manual2.txt
…
```

<payload_size>: Message payload size (in bytes). As explained, the utility not only packages all S1V30120 data files into a single C-array, but it also provides indexing data structures that ease partitioning of the file into message payloads for streaming via the Host CPU serial link. In order to do so, we must specify the size of the payloads that will be encapsulated in each streaming message; once this is done, the output data structures will partition the output data array using this payload size, and provide references to each payload within a file. It is recommended that this parameter is set to 2048, as this will partition each file into 2048 byte payloads (the maximum allowed size for a message via the Host-CPU serial link, see S1V30120 Message Protocol Specification document, section 3.2.2), which will result in file streaming with minimum communication overhead. In any case, *the minimum payload size allowed by the tool is 2 Bytes*.

As a example, assuming that our input file list is called filelist.txt, and that messages of size 2048 bytes are possible (minimum communication overhead) we would invoke the data packaging utility as follows:

```
S1V30120_data_pack.exe ../data/filelist.txt 2048
```

5.2.2 S1V30120 Data Packaging tool Output Files

Fig. 9 shows the output files generated by the S1v30120 Data Packaging Utility:

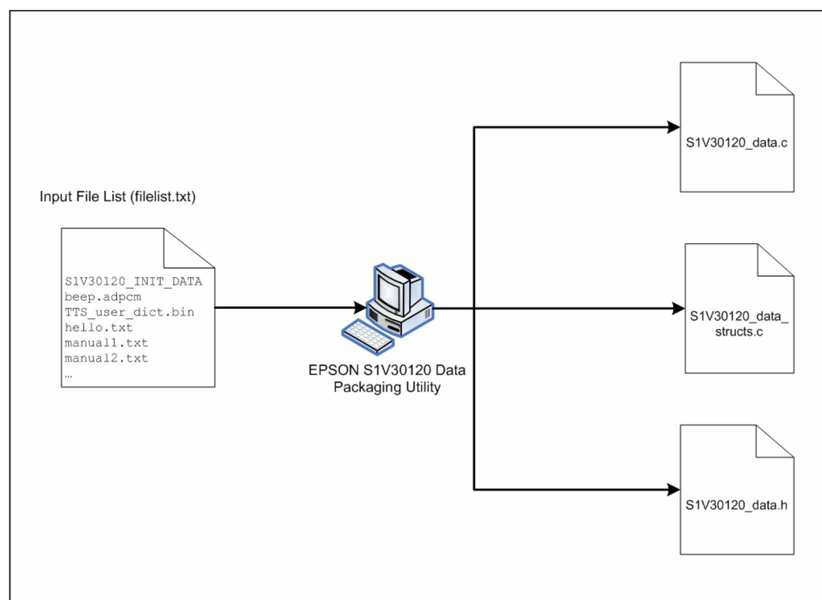

Fig. 9 Output files of S1V30120 Data Packaging Utility

As can be seen, the following output files are created:

- **S1V30120_data.c**: it contains a constant byte array with the data from all input files concatenated in it:

```
/* S1V30120 Voice Guidance chip system data
/* Concatenated into a single array by S1V30120_data_pack utility*/
#include "typedefs.h"

const uWord8 S1V30120_data[93428] =
{ 0x10,0x54,0x78,0x54,0x12,0x68,0x10,0x54,0x78,0x130x54,0x12,0x68,
  0x54,0x78,0x130x54,0x12,0x68,0x54,0x78,0x54, 0x12,0x68, 0x10,
  0x68,0x10,0x54,0x78,0x130x54,0x12,0x68, ,0x54,0x78,0x54,0x12,
  0x54,0x78,0x130x54,0x12,0x68,0x54,0x78,0x54, 0x12,0x68, 0x10,
  0x54,0x78,0x130x54,0x12,0x68,0x54,0x78,0x54, 0x12,0x68, 0x10
  . . .
  . . .
  0x98, 0x32, 0x12};
```

Fig. 10 Example of the contents of the output file S1V30120_data.c

Note that the type of the array is "const uWord8". uWord8 is a generic type indicating "unsigned 8-bit word" that has to be translated to the equivalent Host CPU native type in the typedef.h header file. In a typical 32-bit CPU, such header file would look as follows (although internal type representation varies with each architecture, consult the development tool-chain documentation of your Host CPU):

```
/* typedefs.h: machine-independent type definitions*/
#ifndef NULL
        #define NULL (void *) 0
#endif

/* Mapping of machine types into generic types */

typedef char Word8;
typedef short Word16;
typedef int Word32;
typedef unsigned char  uWord8;
typedef unsigned short uWord16;
typedef unsigned int   uWord32;
```

Source Code 1 Mapping machine-independent integral types to CPU dependent types: an example of typedef.h

- **S1V30120_data_structs.c**: this file contains indexing and access data structures that make it straightforward to split files into payloads for messages to be streamed via the SPI link. The data structures are organized as follows:

First, the utility calculates how many payloads are required to stream each of the files in the data array, and creates an array of pointers to the beginning of each payload and an array of 16 bit words storing the length of each payload; this is carried out for each input file. The process is depicted for the example input `filetext.txt` in the next figure:

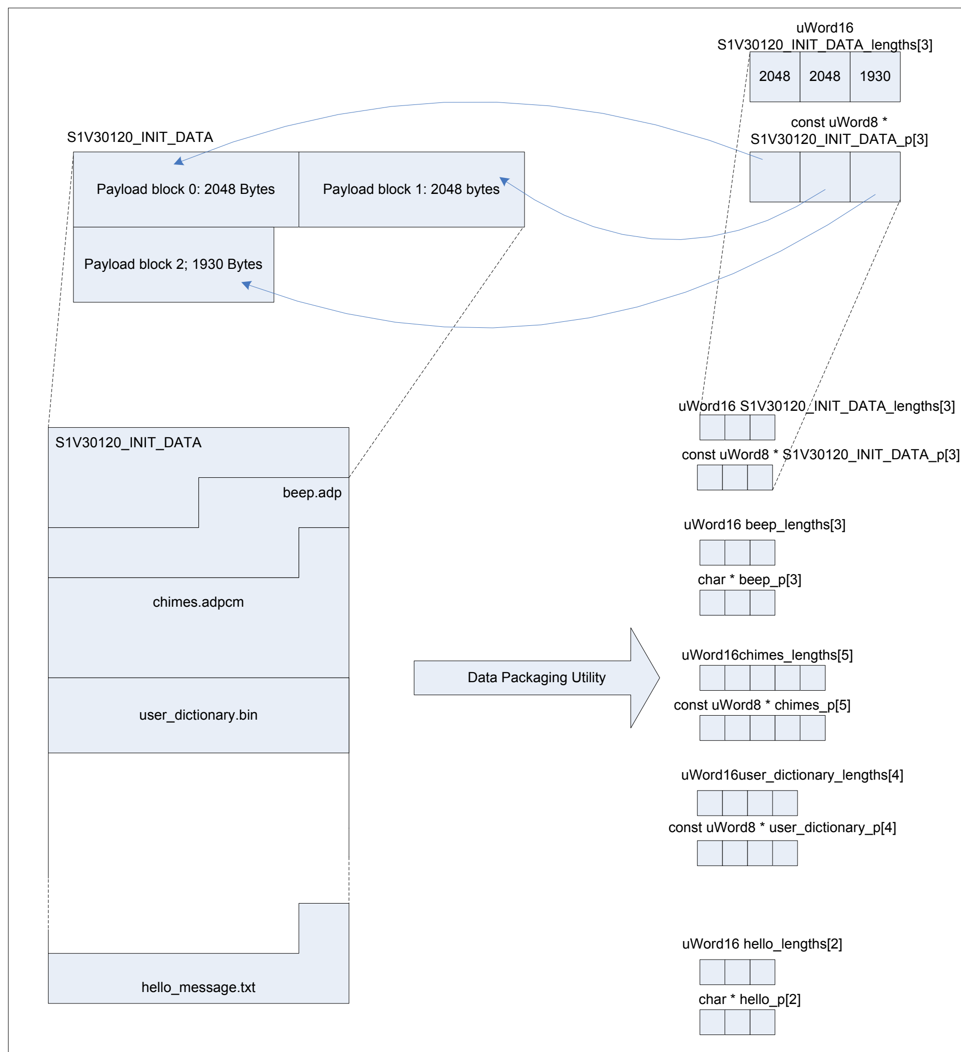

Fig. 11 Message indexing data structures created by the S1V30120 Data Packaging Utility

It is clear that in order to stream a file (i.e. to send a sequence of messages each one with one of the payloads belonging to a given file), all that is required is a reference to the *payload pointer* and the *payload length* arrays of that file.

After this, the utility creates a set of higher level indexing arrays providing a way to index and reference the payload and length arrays for each file. Essentially the following structures are created:

- **`uWord8 * s1v30120_filenames[S1V30120_N_FILES]:`** a table containing all the file names in the input file list.

- `uWord8 ** s1v30120_payloads[S1V30120_N_FILES`]: a set of references to the payload pointer array for each file

- `uWord16 * s1v30120_payload lengths[S1V30120_N_FILES`]: an array containing references to the payload length array for each file

Fig. 13 depicts these data structures:

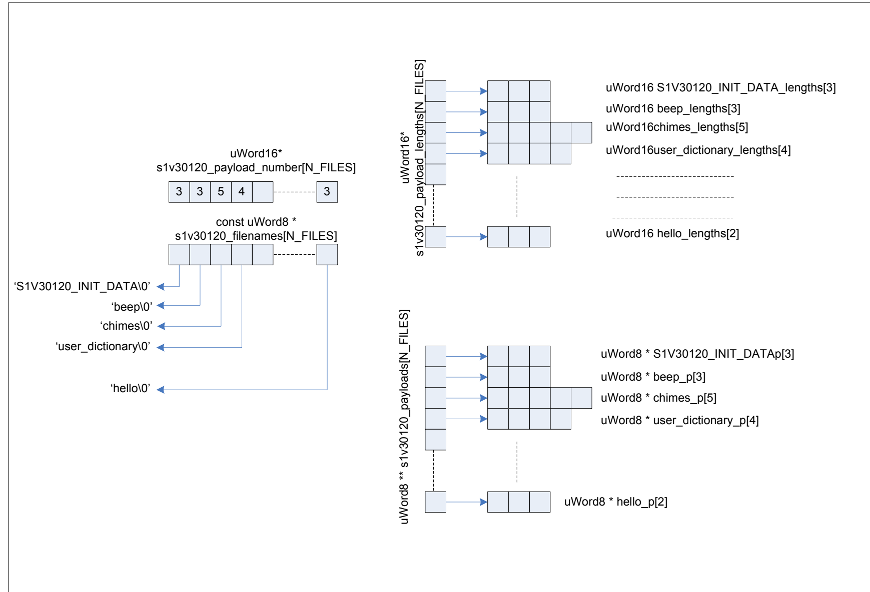

Fig. 12 File indexing data structures created by the S1V30120 Data Packaging Utility

Note that the ordering in all indexing arrays is consistent: if the index for a given filename is **`i`** , then a reference to its payload array can be found in **`s1v30120_payloads[i]`**, and a reference to its payload length array will be in **`s1v30120_payload_lengths[i]`**. Therefore the procedure for accessing a specific file and streaming it in messages is straightforward:

1. Find the index of the file name in the **`s1v30120_filenames[]`** array.
2. Using this index, find the number of message payloads required to stream the file, in **`s1v30120_payload_number[]`** array.
3. Get a reference to payload data and payload length of the file by accessing **`s1v30120_payload_lengths[]`** and **`s1v30120_payloads[]`** arrays with the file name index.

Once a reference to payload data and length arrays is obtained, we have an array of pointers to each data message payload, and an array with the length of each of those payloads, with the actual number of payloads that the message is split into. This is all we require to stream the message.

- `S1V30120_data.h`: This file contains definitions required for the previous files:

```
/*  s1v30120_data.h : header for S1v30120_data files
    Generated automatically by EPSON S1V30120 Data Packaging Utility*/

#define S1V30120_N_FILES 6
#define S1V30120_BASE_ADDR (&s1v30120_data[0])
```

Source Code 2 data header generated by the S1V30120 Data Packaging Utility

5.2.3 Accessing S1V30120 data: an example API

Once the S1V30120 data files have been packaged with the data packaging utility, the data files have become effectively source code (a byte array). Therefore integrating it into your system should be as simple as including the generated files into the Host CPU firmware project, and specifying the address where this data is to be place in the Host CPU linker script. However, there is still considerable development work to be done in terms of pointer dereferencing and house-keeping for accessing the S1V30120 data.

In order to ease this task, an example of how to implement an API for data access is presented: the stream API. The functionality of the stream API is based on the type **`stream_t`**, a structure of contents transparent to the application client. Once a variable of type **`stream_t`** is bound to a S1V30120 data file, the application running on the host CPU is not required to directly handle pointer de-referencing while streaming a file via the host interface: the stream API automatically delivers references to the next payload to be encapsulated in a message, without the Host CPU having any awareness of the details. The API contains the following functions:

`Word16 stream_reset( stream_t * stream`) - initializes an stream structure

Parameters:

- **`stream_t`** stream: the stream to be initialized

`Word16 stream_bind(stream_t * stream, uWord8 * filename`) - binds a stream structure to a given file in the S1V30120 data structures. Once bound, the data in the file can be accessed by using the stream API only.

Parameters:

- **`stream_t stream`** : the stream to be bound
- **`uWord8 * filename`** : an array of characters with the file name

Returns:

0 if the stream was bound successfully

-1 if there was an error (the file doesn't exist)

`uWord8 * stream_get_payload(stream_t * stream):` it retrieves a payload from the file being streamed. Subsequent calls to this function retrieve the sequence of payloads belonging to the bound file. In other words, the stream keeps track of it own state, iterating through the payload array with each call to stream_get_payload.

Parameters :

- **`stream_t * stream`**: reference to the stream currently in use

Returns:

- **`uWord8`** pointer to the next payload to be streamed if successful
- **`NULL`** if all the payloads from the bound file have been already retrieved

---

`uWord16 stream_get_length(stream_t * stream) :` it returns the length of the current payload to be streamed via the Host CPU serial interface. It worth noting that by *current*, we mean the payload to be retrieved in the next stream_get_payload call. Therefore, in order to obtain all the information required to stream a payload (i.e a pointer to it and its length), we must first get its length, and then the payload itself (otherwise we will obtain the length of the next payload, rather than the current one).

Parameters :

- **`stream_t * stream`**: reference to the stream in use

Returns :

length of the current stream if successful

0 if all the payloads in the stream have been already consumed.

The following code shows the implementation of the stream API shown above. The API requires the files created by the EPSON Data Packaging tool and **`typedef.h`**. Any client code accessing this API has to include the **`stream.h`** header.

Source Code 3 – stream.h

```
/*-------------------------------------------------------------------------+
 | stream.h                                                                |
 +-------------------------------------------------------------------------+
 | An example API for encapsulating the data structures generated by the   |
 | S1V30120 Data Packaging Utility. Use only the functions in this header. |
 +-------------------------------------------------------------------------*/

#include "typedef.h"  /* For mapping machine types to generic types */

/*-------------------------------------------------------------------------+
 | stream_t type - Structure holding all the required information to access |
 |              and stream data from S1V30120 files.                        |
 +-------------------------------------------------------------------------+
 | Struct Members:                                                          |
 | - payload: Pointer to array of pointers addressing each of the message  |
 |          payloads in the file.                                           |
 | - payload_length: Pointer to the array of payload lengths in the file.  |
 | - streamID: Identifies an entry in the S1V30120 filenames list.         |
 | - payloads_number: The number of messages required to completely stream |
 |                  the file from Host CPU to S1V30120.                     |
 | - payload_index: Keeps track of the current payload/length to be delivered |
 |                 if the stream is queried with steam_get_payload() or    |
 |                 stream_get_length().                                     |
 +-------------------------------------------------------------------------*/
typedef struct stream_t
{
   uWord16* payload_length;
   uWord8** payload;
   Word16 streamID;
   uWord16 payloads_number;
   uWord16 payload_index;
} stream_t;

/*-------------------------------------------------------------------------+
 | stream_bind() - Binds a stream data structure to a S1V30120 data file.   |
 +-------------------------------------------------------------------------+
 | Parameters:                                                              |
 | - filename: The name of the S1V30120 data file to be bound.             |
 |   - stream: The structure which will hold the payloads-lengths data.    |
 |   Returns: 0 if successful, -1 if file name cannot be found.            |
 +-------------------------------------------------------------------------*/
Word16 stream_bind(uWord8* filename, stream_t* stream);

/*-------------------------------------------------------------------------+
 | stream_reset() - Resets (unbinds) a stream.                              |
 +-------------------------------------------------------------------------+
 | Parameters:                                                              |
 | - stream: Pointer to the stream to be unbound.                           |
 |  Returns: 0                                                              |
 +-------------------------------------------------------------------------*/
Word16 stream_reset(stream_t* stream);

/*-------------------------------------------------------------------------+
 | stream_get_payload() - Provides a payload to be sent in a message.       |
 +-------------------------------------------------------------------------+
 | Parameters:                                                              |
 | - stream: Pointer to the stream from which payloads are extracted.      |
 |  Returns: A pointer to the next payload in the stream if successful or, |
 |          NULL if the end of the stream has been reached.                 |
 +-------------------------------------------------------------------------*/
uWord8* stream_get_payload(stream_t* stream);

/*-------------------------------------------------------------------------+
 | stream_get_length() - Provides the length of the current payload.        |
 +-------------------------------------------------------------------------+
 | Parameters:                                                              |
 | - stream: Pointer to the stream from which payloads are extracted.      |
 |  Returns: The length of the current payload or, 0 if the end of stream has |
 |          been reached.                                                   |
 +-------------------------------------------------------------------------*/
uWord16 stream_get_length(stream_t* stream);

/* EOF - stream.h */
```

Source Code 4 – stream.c

```
/*---------------------------------------------------------------------------+
 | stream.c                                                                  |
 +---------------------------------------------------------------------------+
 | An example API for encapsulating the data structures generated by the     |
 | S1V30120 Data Packaging Utility                                           |
 +---------------------------------------------------------------------------*/

#include "typedef.h"      /* For mapping machine types to generic types  */
#include "s1v30120_data.h"  /* header generated by the Data Packing Utility */
#include "stream.h"        /* For streaming functionality               */

/* declarations of data structures created by the utility */
extern uWord8** s1v30120_payloads[S1V30120_N_FILES];
extern uWord16* s1v30120_payload_lengths[S1V30120_N_FILES];
extern uWord8*  s1v30120_filenames[S1V30120_N_FILES];
extern uWord16 s1v30120_payloads_number[S1V30120_N_FILES];

/* Internal help funtions, for internal API use only*/
Word16 string_match(uWord8* string1, uWord8* string2);
Word16 stream_getID(uWord8* filename);

/*---------------------------------------------------------------------------+
 | stream_bind() - binds a stream data structure to a S1V30120 data file      |
 +---------------------------------------------------------------------------+
 | Parameters:                                                               |
 | - filename : The name of the S1V30120 data file to be bound.               |
 | - stream   : The structure which will hold the payloads-lengths data.      |
 |                                                                            |
 | Returns: 0 if successful, -1 if file name cannot be found                  |
 +---------------------------------------------------------------------------*/

Word16 stream_bind(uWord8* filename, stream_t* stream)
{
   /* Get the stream ID for the specified filename */
   stream->streamID = stream_getID(filename);

   /* Check whether the filename was found in the list of s1v30120 files */
   if (stream->streamID < 0)
      return -1;

   /* Set up the stream settings according to the filename specified */
   stream->payload        = s1v30120_payloads[stream->streamID];
   stream->payload_length  = s1v30120_payload_lengths[stream->streamID];
   stream->payloads_number = s1v30120_payloads_number[stream->streamID];
   stream->payload_index   = 0;

   return 0;
}

/*---------------------------------------------------------------------------+
 | stream_reset() - Resets (unbinds) a stream.                                |
 +---------------------------------------------------------------------------+
 | Parameters:                                                               |
 | - stream : Pointer to the stream to be unbound.                           |
 |   Returns: 0                                                              |
 +---------------------------------------------------------------------------*/
Word16 stream_reset(stream_t* stream)
{
   stream->payload        = NULL;
   stream->streamID       = -1;
   stream->payload_length  = NULL;
   stream->payloads_number = 0;
   stream->payload_index   = 0;

   return 0;
}
```

stream.c (continued)

```
/*--------------------------------------------------------------------------+
 | stream_getID() - Get the ID (the position in the S1V30120 file name list)  |
 |              of a file.                                                    |
 +--------------------------------------------------------------------------+
 | Parameters:                                                               |
 | - filename : The name of the S1V30120 data file to be found.              |
 |    Returns: 0 if found, -1 if filename not in list.                       |
 +--------------------------------------------------------------------------*/
Word16 stream_getID(uWord8* filename)
{
   uWord16 i;

   /* Check each file in our list against the given name, if a match is found
      return its ID */
   for (i = 0; i < S1V30120_N_FILES; i++)
      if (string_match(filename, s1v30120_filenames[i]))
         return i;

   /* If no file in the S1V30120 file list matched, return error */
   return -1;
}

/* Simple string compare function, for local API use only */
Word16 string_match(uWord8* string1, uWord8* string2)
{
   int i = 0;

   /* Step through letters of the string whilst both are equal, if the end is
      reached the strings must be equal */
   while(string1[i] == string2[i])
   {
      /* Return success if we've reached the end of the string */
      if (string1[i] == '\0')
         return 1;

      /* Point to the next letter in the string */
      i++;
   }

   /* We found a difference before reaching the end of the string, return that
      the two strings are unequal */
   return 0;
}

/*--------------------------------------------------------------------------+
 | stream_get_payload() - Provides a payload to be sent in a message.         |
 +--------------------------------------------------------------------------+
 | Parameters:                                                                |
 |   stream : Pointer to stream from which payloads are extracted.            |
 |  Returns: A pointer to the next payload in the stream if successful or,    |
 |          NULL if the end of the stream has been reached.                   |
 +--------------------------------------------------------------------------*/
uWord8* stream_get_payload(stream_t* stream)
{
   uWord8* payload;

   /* Check to see if the end of the stream has been reached */
   if (stream->payload_index == stream->payloads_number)
   {
      /* return NULL to signal that the end of the stream has been reached */
      return NULL;
   }
   else
   {
      /* Get the pointer to the next payload in stream, then set the index to
         the next payload in the stream */
      payload = *(stream->payload + stream->payload_index);
      stream->payload_index++;

      /* return a pointer to the next payload in the stream */
      return payload;
   }
}
```

stream.c (continued)

```
/*--------------------------------------------------------------------------+
 | stream_get_length() - Provides the length of the current payload.        |
 +--------------------------------------------------------------------------+
 | Parameters:                                                              |
 |    stream : Pointer to stream from which payloads are extracted.         |
 |  Returns: The length of the current payload or, 0 if the end of stream   |
 |           has been reached.                                              |
 +--------------------------------------------------------------------------*/
uWord16 stream_get_length(stream_t* stream)
{
   /* If we have reached the end of the stream, return 0 to signal so */
   if (stream->payload_index == stream->payloads_number)
      return 0;

   /* Return the length of the current payload */
   return *(stream->payload_length + stream->payload_index);
}


/* EOF - stream.c */
```

6. Developing driver code for S1V30120: a simple TTS streaming driver

6.1 Introduction

Once data is packaged and structured in a suitable way, the next step is to develop Host CPU code to stream this data into the S1V30120 as required. In this section we show an example of how a S1V30120 driver could be implemented for a very simple TTS streaming application. Note that this is a simplistic example, not a full implementation. Specifically the following points are assumed:

- The example does not include handling of asynchronous events, possibly generated by the system's user (e.g. pause, volume up/down etc).
- A very simple SPI hardware link is assumed, with no provision for DMA data transfer or buffering. The API assumed is explained in section 6.2.
- It is assumed that the S1V30120 is already initialized (i.e. the boot procedure, as described in the S1V30120 Message Protocol Specification document, section 4, has already taken place). Note that implementing the initialization procedure should be straightforward,as it can be realized by implementing an initialization file streaming handler based on the TTS streaming handler described here.

For full sample code for a complete implementation of file streaming in a typical 32-bit CPU (EPSON C33 CPU family) consult the sample code provided in this evaluation kit release.

6.2 Example S1V30120 driver architecture

A possible design for the driver module architecture is shown in the next figure:

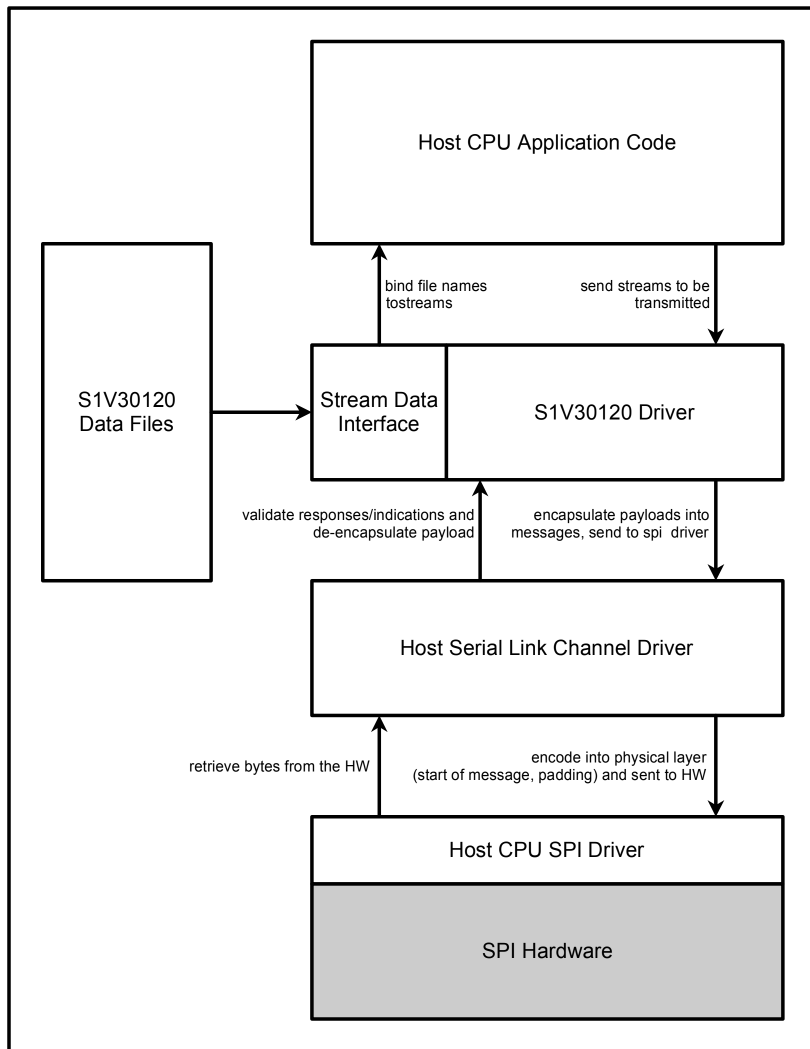

Fig. 13 An example of S1V30120 driver organization

Where these are the main modules:

Host CPU Application module: the application client of the S1V30120 driver. When a TTS file is streamed, the Host Application binds the file name to a stream, and requests file streaming to the S1V30120 module.

Stream Data Interface: this is essentially the stream API described in section 5. Its role is dual: in the first instance, it allows the Host CPU Application Code to binding files to streams. Secondly, it is used by the S1V30120 driver module to iterate through the sequence of file payloads (using the `stream_get_payload()` function).

Host Serial Link Channel Driver: the channel driver sends and receives data from the SPI driver, and handles all encapsulation/de-encapsulation of payloads according to the message protocol. This involves:

- Including Start-Of-Message (0xAA) symbols.
- Encapsulating/de-encapsulating message types and message lengths
- Including padding after transmission/reception

Once this is done, its job is to notify the S1V30120 driver Event Handler of message reception/transmission. Note that the channel driver is effectively the interrupt handling module for the SPI hardware.

Host CPU SPI Driver: the actual driver of the SPI hardware. As such, it is platform dependent and not described here. The only assumption made about the SPI hardware is that it is capable of generating interrupts when a byte is transmitted/received.

S1V30120 Driver: this module is the core of the software modules depicted in figure 13. It is basically a state machine that carries out the sequence of configuration and file streaming messages required to produce speech from a text file. As this is a simple example, we will assume that the chip is already initialized by the time TTS streaming is required. Later, it will become apparent that streaming initialization files can be implemented in a very similar fashion to TTS streaming. There is therefore no advantage to be had by including it in this simple example. Refer to the C33 Host CPU sample code included in this evaluation kit release for actual sample code for the initialization sequence.

6.3 Developing Message Rx/Tx Handlers for the Host CPU: an example SPI Channel driver

The channel driver functionality is essentially the interrupt handler for the SPI hardware in the system. It could be called for each byte received or transmitted and could be built upon two key components:

- A global data state structure, which holds all the required information between interrupt calls.
- The SPI interrupt handling function itself, which calls both the receive and transmit channel handlers. At its core, both Rx/Tx channel handlers could be simple state machines that implement the packet encapsulation required by the message protocol (Start of message (0xAA), Message Length field, Message ID, payload, padding). The following diagrams show how these state machines could be implemented:

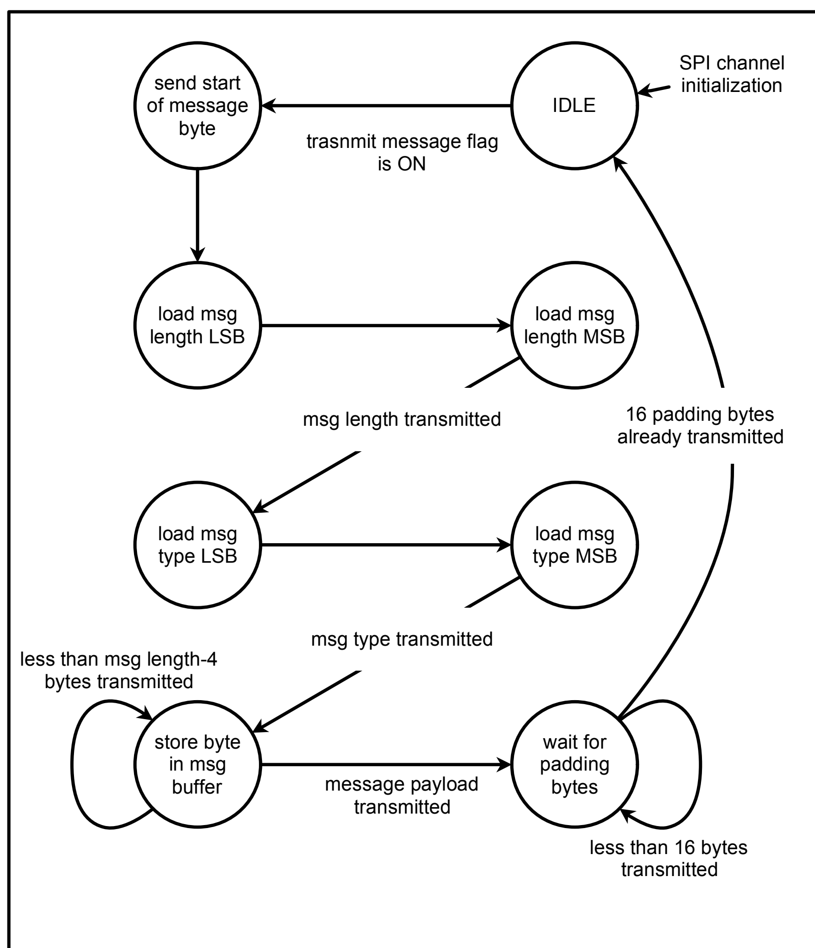

Fig. 14 Behavior of the transmit channel at the SPI channel driver

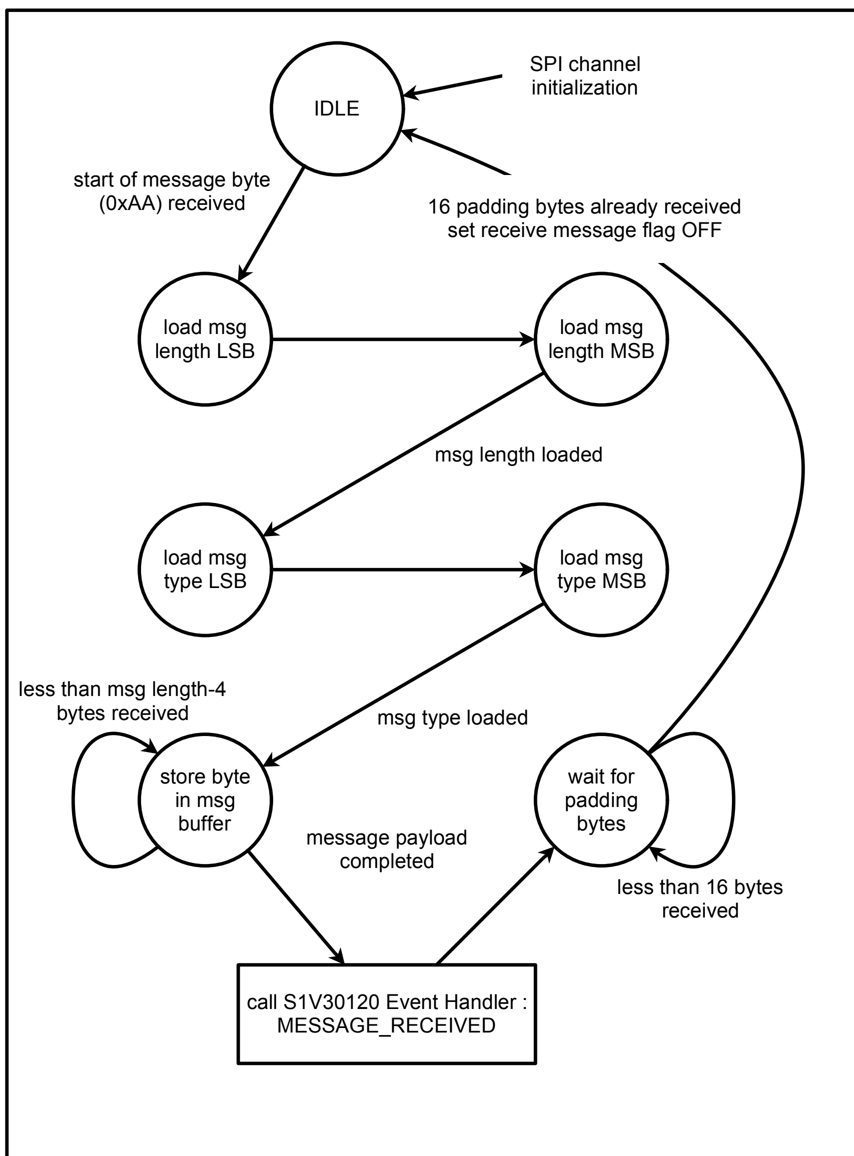

Fig. 15 Behavior of the receive channel at the SPI channel driver

It is worth noting that the link between the SPI channel driver and the S1V30120 driver is done explicitly at the receive channel handler state machine. While a message request from Host CPU to the S1V30120 is initiated by the S1V30120 driver module calling the spi channel to send a message, at the receiver side, the reception of a response message from S1V30120 to Host CPU is signaled to the S1V30120 driver. This signaling is done by sending a call to its event handler when the Rx channel state machines identifies an end of message condition. Furthermore, the start of SPI hardware reception is triggered by a MSG_READY GPIO transition from logic level low to high; therefore a MSG_READY event should also trigger the S1V30120 event handler.

6.4 Streaming Data from Host CPU to S1V30120: an example S1V30120 event handler for TTS streaming

With all the supporting functionality in place, the next step would be to implement a top level driver that handles events to drive the S1V30120.Most of the functionality of the driver could be in fact a simple state machine implemented by the S1V30120 event handler, and triggered by events generated either by the MSG_READY line, the SPI receive channel, or the application firmware itself. Fig. 16 shows the internals of such state machine, which essentially follows the flow for TTS streaming described in figure 24 of the S1V30120 Message Protocol Specification document:

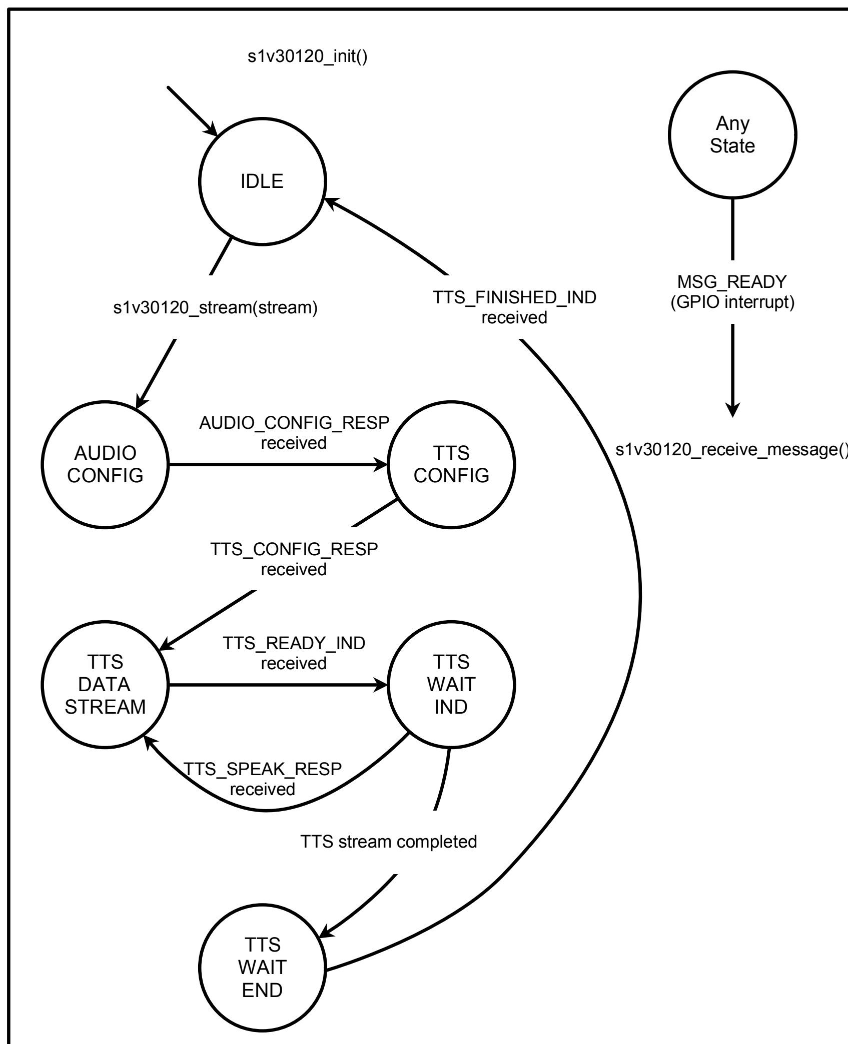

Fig. 16 S1V30120 Event Handler State Machine

6.5 Developing for low performance Host CPUs : Half Duplex Operation

The S1V30120 offers a very flexible message interface that allows for handling of asynchronous events, such as:

- Pausing/Stopping the TTS/ADPCM playback at any time
- Modify the output volume at the S1V30120 Digital to Analogue Converter
- S1V30120 GPIO output control from Host CPU

This asynchronous event handling might be difficult to support for Host CPUs with very limited resources (such as 8-bit MCUs or CPUs running at low clock rate). To aid integration in systems with such constraints, it is possible to control the S1V30120 in such a way that the communication between it and the Host CPU is half-duplex. This is possible by severely constraining the handling and timing of asynchronous requests. By following the constraints below, it is guaranteed that the S1V30120 will not generate responses/indications while the Host CPU is sending a request message:

1. Do not use Pause playback messages: ISC_TTS_PAUSE_REQ, ISC_SPCODEC_PAUSE_REQ messages are not safe for half-duplex communication (as they can collide with indications from the S1V30120).
2. Use ISC_TTS_STOP_REQ message ONLY when a ISC_TTS_FINISHED_IND is received (in other words, once TTS playback is started, it cannot be stopped, or half-duplex communication cannot be guaranteed).
3. Do not send any audio/GPIO related message while TTS/ADPCM playback is in operation. For volume changes/GPIO output events, wait for the receipt of an ISC_TTS_FINISHED_IND or ISC_SPCODEC_FINISHED_IND.
4. Make sure that no corrupted file/data is sent to the S1V30120 and that no unexpected message is sent from the Host CPU. This will generate a ISC_ERROR_IND fatal error that can take place at any time (including during a Host CPU request message).
5. By following guidelines 1-4 it is guaranteed that no response/indication message will be sent from the S1V30120 while the Host CPU is sending a request message. However, it is possible that during the sending of padding bytes from the Host CPU after a request, the S1V30120 signals a response/indication by asserting its MSG_READY line. For this reason, it is necessary for the Host CPU to poll the MSG_READY line while sending padding after a request message.

Although these guidelines limit severely asynchronous event handling, they make possible to ensure half-duplex communication and totally deterministic message flow between the S1V30120 and the Host CPU.

EPSON International Sales Operations

AMERICA

EPSON ELECTRONICS AMERICA, INC.
HEADQUARTERS
2580 Orchard Parkway
San Jose , CA 95131,USA
Phone: +1-800-228-3964 FAX: +1-408-922-0238

SALES OFFICES
Northeast
301 Edgewater Place, Suite 210
Wakefield, MA 01880, U.S.A.
Phone: +1-800-922-7667 FAX: +1-781-246-5443

EUROPE

EPSON EUROPE ELECTRONICS GmbH
HEADQUARTERS
Riesstrasse 15
80992 Munich, GERMANY
Phone: +49-89-14005-0 FAX: +49-89-14005-110

ASIA

EPSON (CHINA) CO., LTD.
23F, Beijing Silver Tower 2# North RD DongSanHuan
ChaoYang District, Beijing, CHINA
Phone: +86-10-6410-6655 FAX: +86-10-6410-7320

SHANGHAI BRANCH
7F, High-Tech Bldg., 900, Yishan Road,
Shanghai 200233, CHINA
Phone: +86-21-5423-5522 FAX: +86-21-5423-5512

EPSON HONG KONG LTD.
20/F., Harbour Centre, 25 Harbour Road
Wanchai, Hong Kong
Phone: +852-2585-4600 FAX: +852-2827-4346
Telex: 65542 EPSCO HX

EPSON Electronic Technology Development (Shenzhen) LTD.
12/F, Dawning Mansion, Keji South 12th Road,
Hi- Tech Park, Shenzhen
Phone: +86-755-2699-3828 FAX: +86-755-2699-3838

EPSON TAIWAN TECHNOLOGY & TRADING LTD.
14F, No. 7, Song Ren Road,
Taipei 110
Phone: +886-2-8786-6688 FAX: +886-2-8786-6660

EPSON SINGAPORE PTE., LTD.
1 HarbourFront Place,
#03-02 HarbourFront Tower One, Singapore 098633
Phone: +65-6586-5500 FAX: +65-6271-3182

SEIKO EPSON CORPORATION
KOREA OFFICE
50F, KLI 63 Bldg., 60 Yoido-dong
Youngdeungpo-Ku, Seoul, 150-763, KOREA
Phone: +82-2-784-6027 FAX: +82-2-767-3677

GUMI OFFICE
2F, Grand B/D, 457-4 Songjeong-dong,
Gumi-City, KOREA
Phone: +82-54-454-6027 FAX: +82-54-454-6093

SEIKO EPSON CORPORATION
SEMICONDUCTOR OPERATIONS DIVISION

IC Sales Dept.
IC International Sales Group
421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN
Phone: +81-42-587-5814 FAX: +81-42-587-5117

Document Code: 411333000
First Issue January 2008