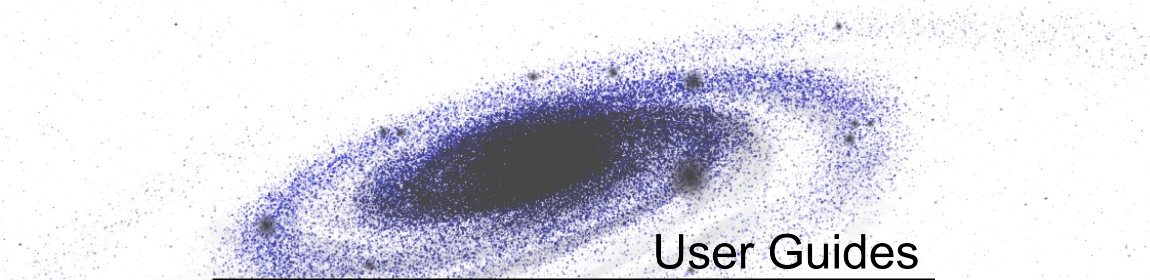
Analyser | PlayTime!

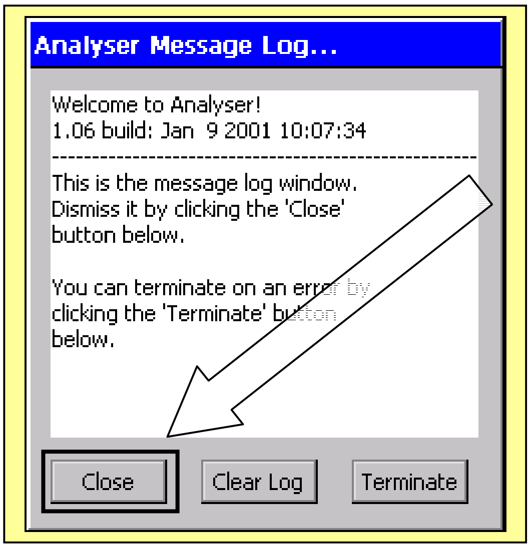
v1.06 | v0.04 alpha



User Guides

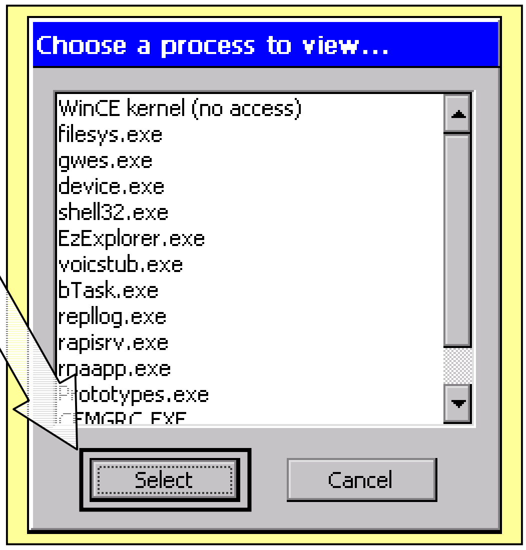
Document version	2.07
Created	15th January, 2001
Last Modified	30th January, 2001
Analyser version	1.06
PlayTime version	0.04a
© 2001 *AdAstra* © 2001 QUT SDRL	Dylan Muir (dr.muir@student.qut.edu.au)

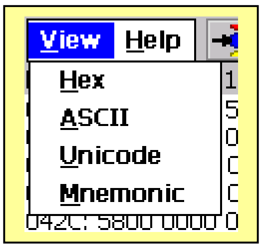
Analyser Quick Guide



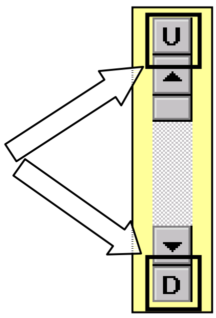
1. Dismiss the welcome message by tapping the "Close" button.

2. Tap the "Choose Process" button on the command bar, and select a process to inspect. Tap the "Select" button to view the process.





3. Use the "View" menu to select a format in which to view the process' data.



4. Use the scroll bar and the up / down buttons to navigate through the process' memory space.

Overview

Analyser and PlayTime together comprise a toolkit for learning about programs and data structures in memory. The two tools run under Windows CE, which provides a portable and accessible environment on which to experiment. Students can inspect and modify all programs on the machine, including portions of the operating system. Because the Windows CE OS is stored in Read-Only Memory, it cannot be corrupted. The system is therefore quite robust, and places no limitations on tinkering.

Analyser enables the data of running processes in memory to be inspected and modified in a variety of formats. It can also disassemble processes to display the source code in SH3 assembly language. It provides tools to locate and analyse data structures, and to navigate through a process' entire address space.

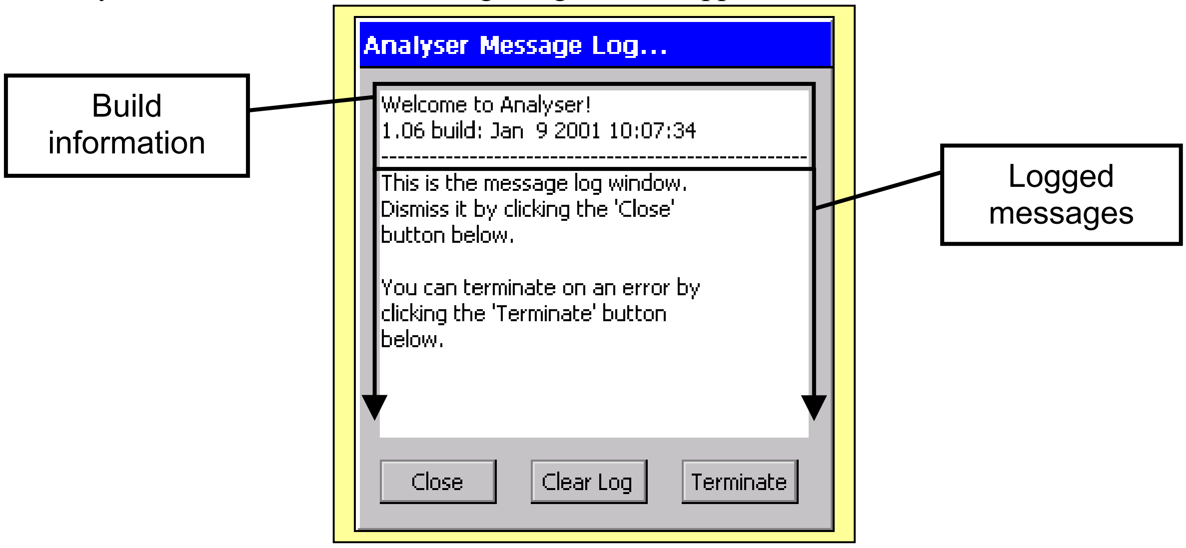
PlayTime is designed as a set of exercises to help the student become familiar with the internal structures of programs. PlayTime provides a number of data structures that are easily inspected and modified by Analyser. Students will learn how to decode basic data structures such as text, integers, floating point numbers and arrays. PlayTime also allows a section of its code to be modified and executed. The results of these changes can be seen through a number of dedicated output fields accessed from the user interface of PlayTime itself.

Table of Contents

Analyser user guide

Message Log window

When Analyser is first loaded, the Message Log window appears with some basic information.

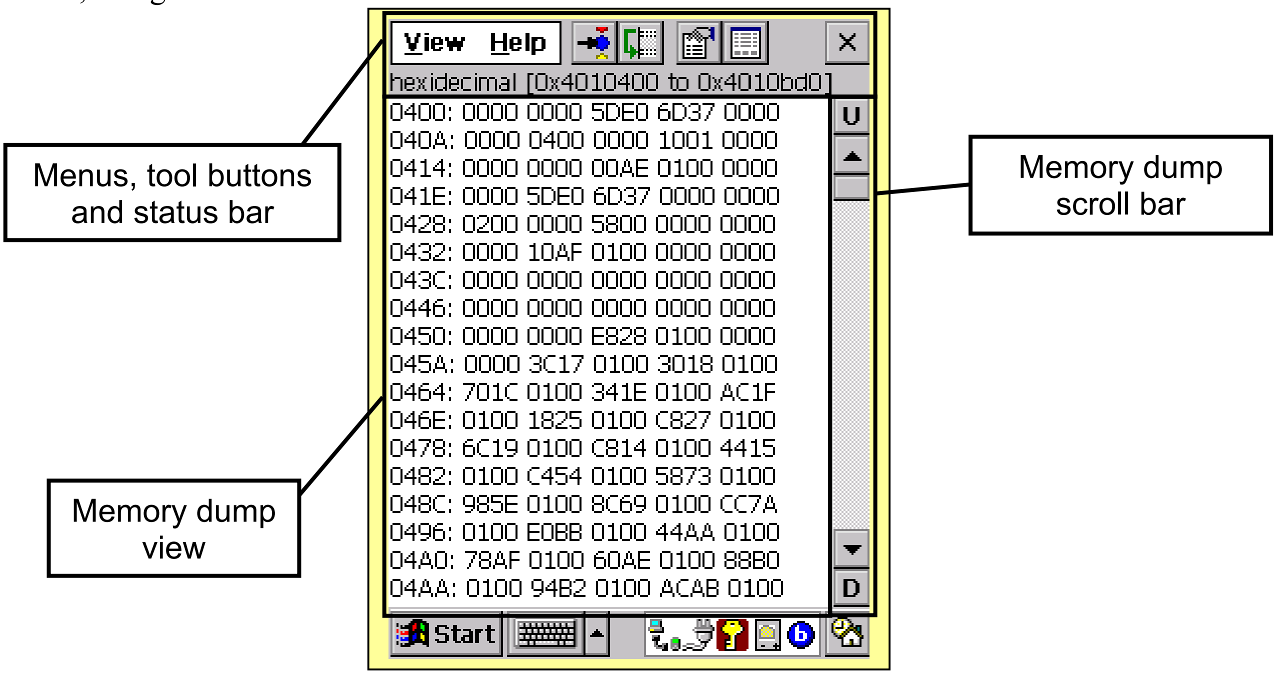


The Message Log keeps a record of all messages and information displayed to the user in an Analyser session. The Message Log is also used to display process information and help.

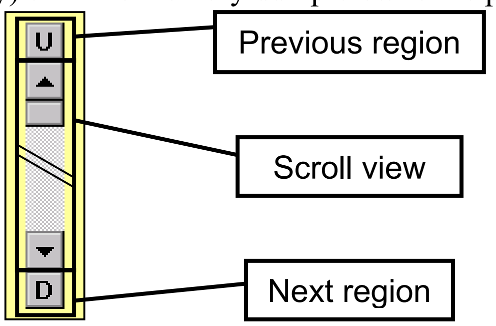
To close the window and return to the main Analyser window, tap the "Close" button.
To clear the current log, tap the "Clear Log" button.
To terminate Analyser on receiving an error, tap the "Terminate" button.

Main window & commands

Analyser is primarily a program for inspecting and modifying processes in memory. The majority of the main display is taken up by the memory dump view. This is a list that displays data residing in memory in various formats. To the right of this list is the scroll bar by which memory can be navigated. At the top of the screen are the menus and tool buttons that control how memory is viewed, along with the status bar.

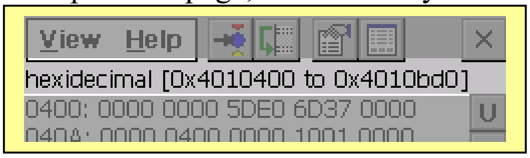


Memory is navigated through a series of regions of data. A single region can contain up to 500 lines of contiguous data in a particular format. The scroll bar next to the memory dump scrolls through this region. The region Up and Down buttons (above and below the scroll bar, respectively) shift the memory dump view to the previous or next region in memory.

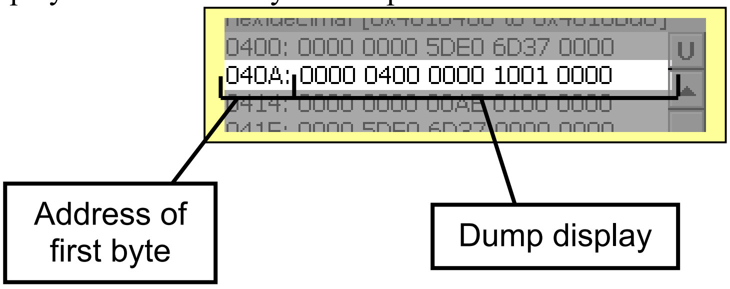


Because of the way a process allocates its storage in memory, not all of the process' data will be in the same area of memory. Analyser will only ever show regions of memory that have been allocated to a process. The next and previous region buttons will move the memory dump view to the next or previous *viewable* areas of memory. This may not be contiguous with the previously viewed region.

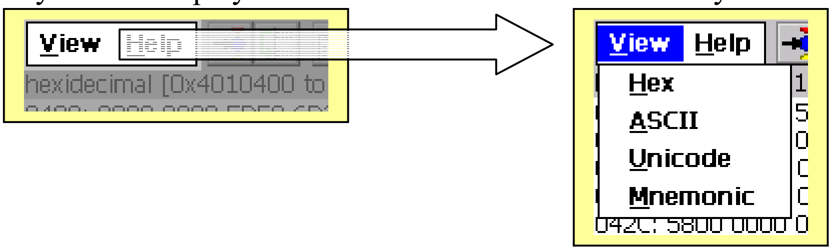
The status bar displays the start and end address of the current region, as well as the format the data is being viewed in. The range addresses are displayed in absolute terms, and indicate the first byte within the region and the last byte of the region. These absolute addresses are seven or eight hex digits (up to 4096 MB). A relative address (obtained from within a process) is based at the beginning of the process' page, and is usually five or six hex digits (can be up to 32 MB).



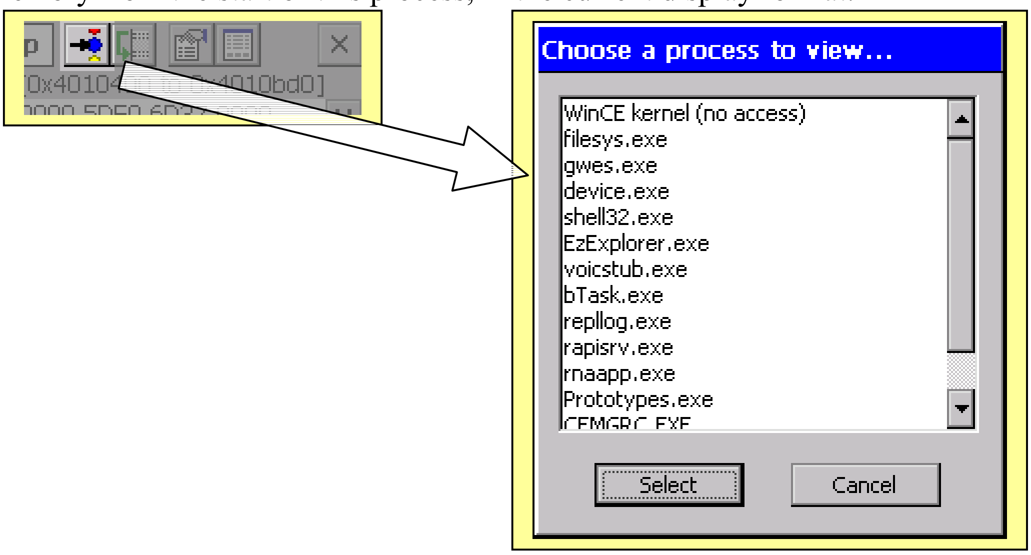
The lines in the memory dump view begin with the abbreviated address of the first byte on the line. The rest of the line displays data in memory in the specified format.



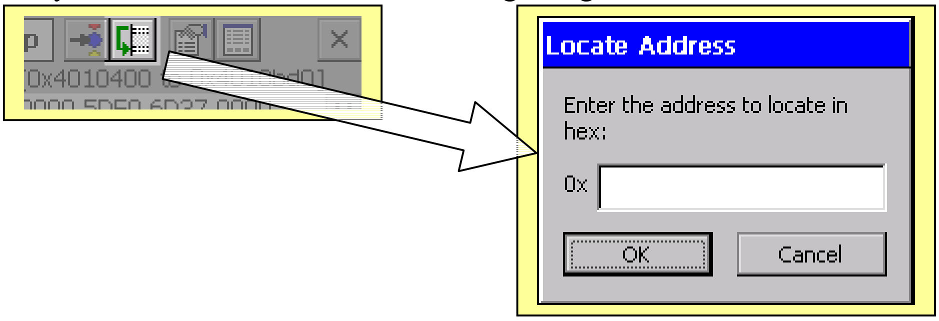
The view menu on the command bar enables the user to change the viewing format of the memory dump. Memory will be displayed in the new format from the first byte of the current region.



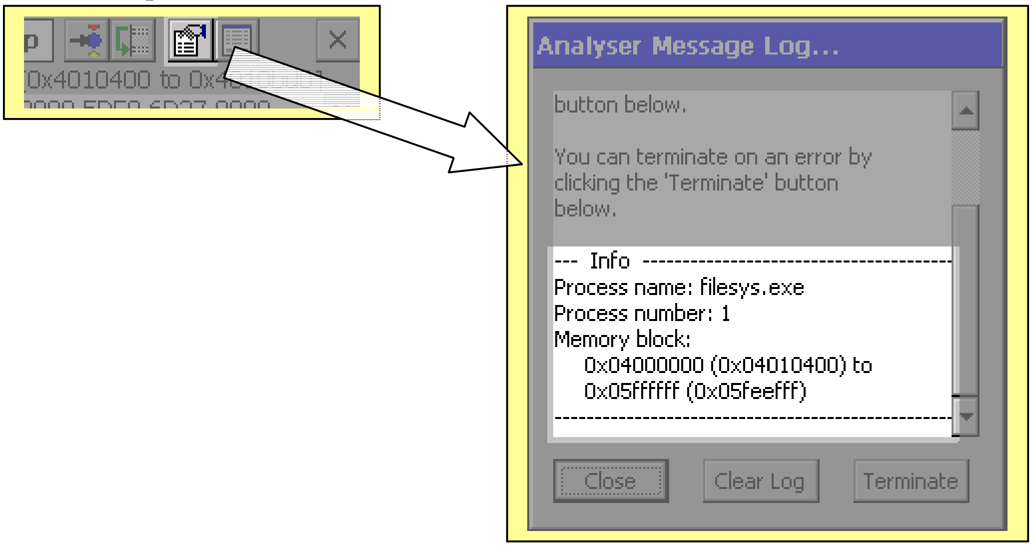
To select a process to view, tap the "Select Process" button on the command bar. A dialog will pop up, listing all currently executing processes. Selecting a process will cause the memory dump view to display memory from the start of this process, in the current display format.



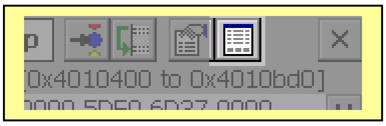
To quickly find a specific address within the current process, tap the "Locate Address" button on the command bar. A dialog will pop up to enable entry of the address. An address entered in this dialog box is assumed to be zero-based at the beginning of the current process' memory page. This enables the entry of relative addresses as well as eight-digit absolute addresses.



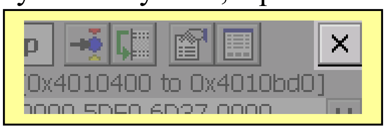
The "Process Info" button on the command bar displays the name of the process and the memory region allocated to the process.



The "Show Message Log" button on the command bar displays the Message Log window, if it is hidden.



To quit Analyser at any time, tap the "Close" button on the command bar.

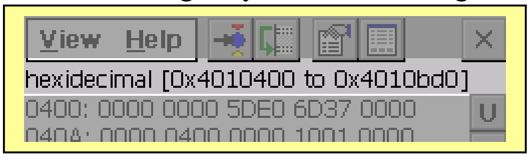


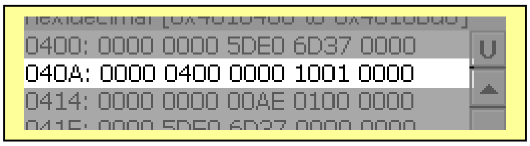
Viewing formats & descriptions

Analyser is capable of interpreting and displaying memory in hexadecimal notation, text (ASCII and Unicode) and disassembled SH3 mnemonics. The View menu switches the display between these formats.

Hex format

The default viewing format for Analyser is hex. Data is displayed byte-by-byte in hexadecimal (base-16) notation. A single byte is two hex digits.





The Hitachi SH3 processor used by the HP Journada uses little-endian addressing. This means that when a word or longer data structure is stored in memory, the least significant byte (LSB) is stored at the lowest address, followed by the next significant in order up to the most significant byte (MSB) at the highest address. In Analyser, the data appears left to right from LSB to MSB. This is intuitive for single byte data, but reads backwards to how we would write a four digit hex word or an eight digit hex double-word. The double-word $\texttt{0x12345678}_{16}$ would appear as "`7856 3412`" in Analyser.

The bytes on a line of hex data are grouped in twos for ease of reading, but are still displayed in sequential order.

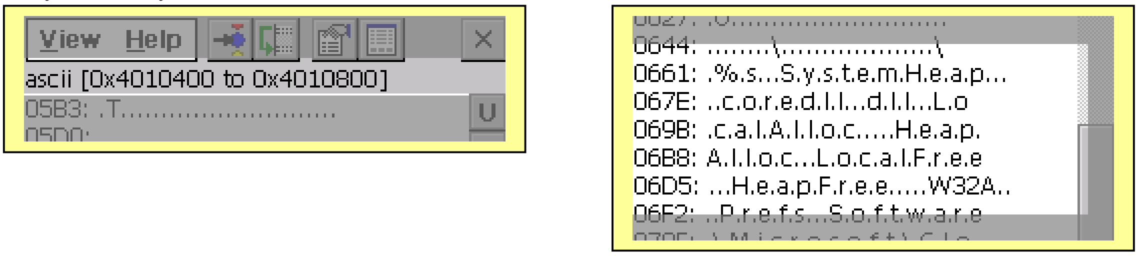
Unicode format

All strings in Windows CE are in Unicode format, which allows extended character sets for Kanji and other scripts. The display in Analyser is filtered to display only printable characters. All other characters are replaced by a full-stop ('.').

Unicode characters are two bytes in length. Printable Unicode characters have the same byte codes as their standard ASCII counterparts, but are padded out with zeros to fill two bytes.

ASCII format

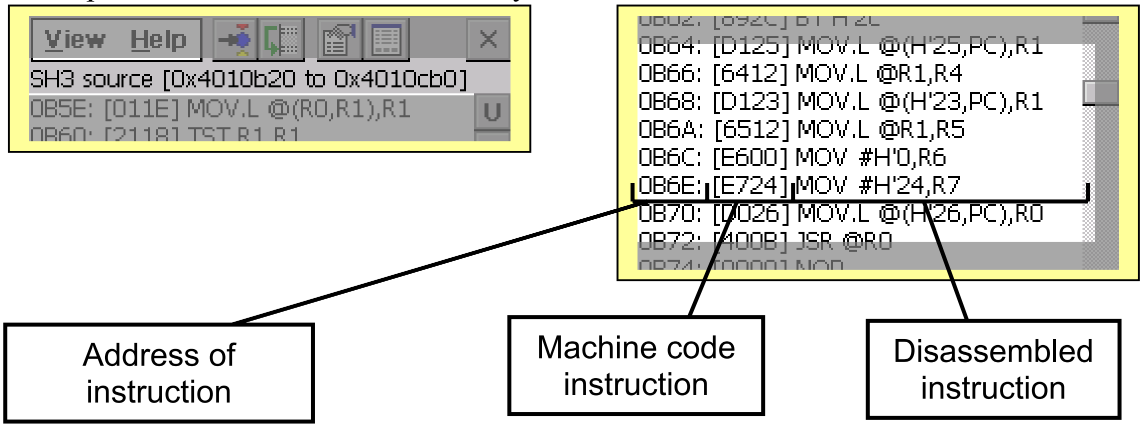
The ASCII viewing format interprets all data as standard C strings. Like Unicode format, Analyser filters the data to display only printable characters. ASCII characters occupy a single byte each. A Unicode string viewed in ASCII format will have a character every second byte and a <nul> (0x00 character) every other byte.



Compare the (ASCII) text shown in the image on the right with the same text in Unicode format on the previous page.

SH3 mnemonics

This format attempts to disassemble data in memory into Hitachi SH3 assembler mnemonics.

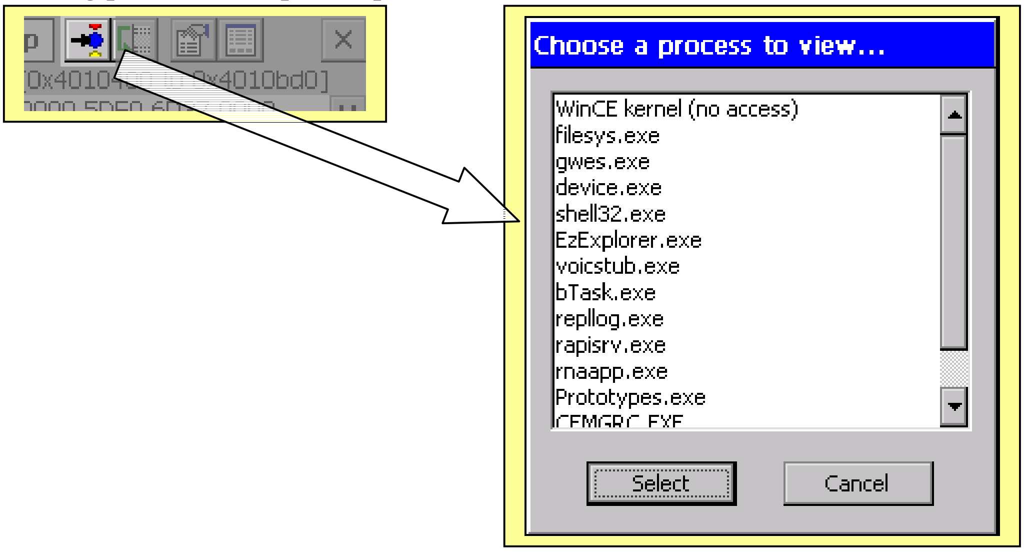


The machine code instructions are shown in square brackets, followed by the disassembled opcode and operands. Invalid instructions are shown by a message, and are a good indication that the area of memory currently being viewed is not code!

The SH3 has a RISC (Reduced Instruction Set Computer) type instruction set, so all machine code instructions fill a word (16 bits). This is in contrast to a traditional instruction set machine such as an Intel x86 processor which has variable-length instructions.

Selecting processes

To select a running process to inspect, tap the “Select Process” button on the command bar.



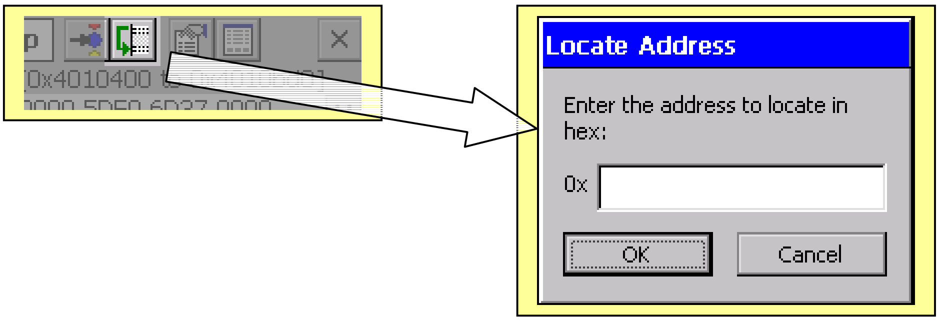
A dialog will pop up, listing all the currently executing processes. The first process in the list is always the Windows CE kernel. This process is protected, and cannot be inspected or modified. All other processes are listed as the name of the executable file where they originated. Choose a process from the list and tap “Select.” The memory dump view will switch to the first region of memory viewable in the process.

The Windows CE operating system consists of five processes, including the kernel.

nk.exe	Contains the Windows CE kernel (protected).
filesys.exe	Manages the persistent object store database and transactions.
gwes.exe	Supports the Win32 system API and windowing system.
device.exe	Manages system devices and device drivers.
shell32.exe	Provides the system shell and user interface (taskbar, etc.).

Locating data

To jump directly to a known address in memory, tap the “Locate Address” button on the command bar. A dialog box will appear to allow the entry of an address to display in the memory dump view.



Processes in Windows CE have a 32MB virtual address space each.

Process 1 (32 MB)	$\texttt{0x02000000}_{16}$ $\texttt{0x03ffffff}_{16}$
Process 2 (32 MB)	$\texttt{0x04000000}_{16}$ $\texttt{0x05ffffff}_{16}$
…	
Process 32 (32 MB)	$\texttt{0x64000000}_{16}$ $\texttt{0x65ffffff}_{16}$

The Microsoft Windows CE documentation has this to say about loading processes into memory:

> When a process initialises, the OS stores in the slot that is assigned to the process all of the dynamic-link libraries (DLLs), the stack, the heap, the application code, and the data section for each process. DLLs are loaded at the top of the slot, followed by the stack, the heap, and the executable file (.exe). The bottom 64 KB is always left free.

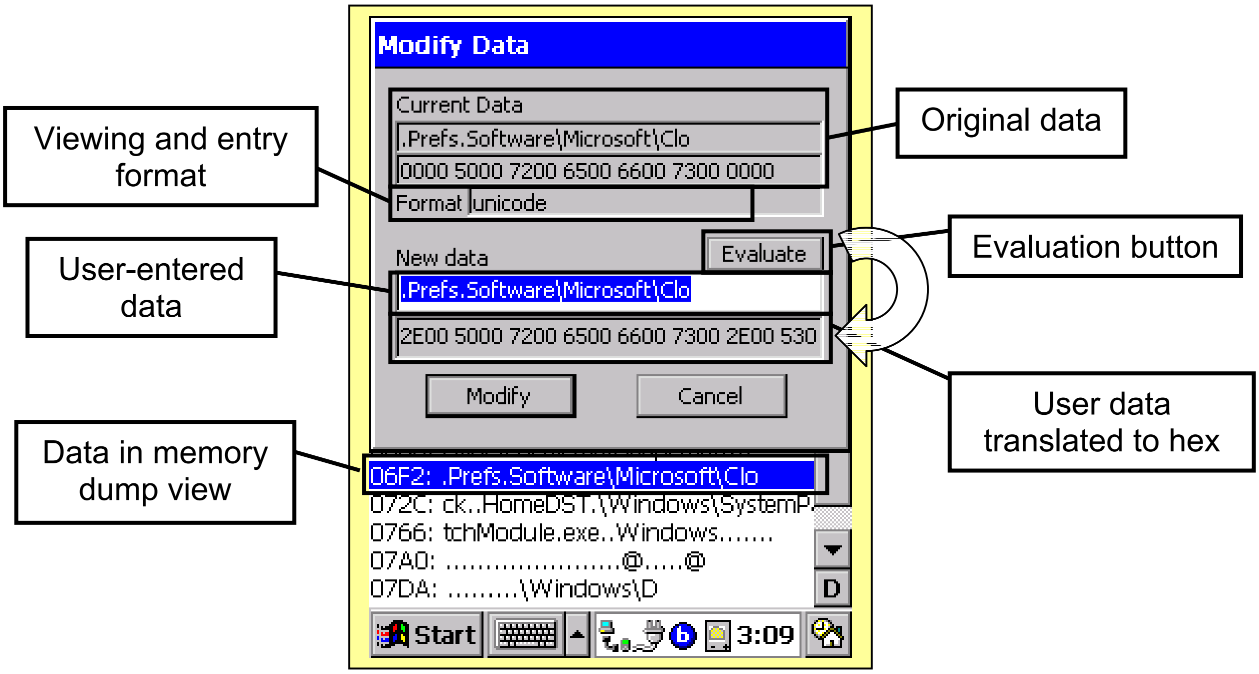
MSDN Library, July 1999

A local (relative) address displayed from within a process is zero-based at the start of the process’ virtual address space. Therefore if process 2 displays an address of $0x1050_{16}$, this address is really $0x04001050_{16}$ in the system’s virtual address space. Analyser understands this, and if you enter a relative address into the locate address dialog box Analyser will compensate for this offset. If you enter an absolute address with the full eight hexadecimal digits, Analyser will understand this, too.

If the address is both within the memory space of the current process and is viewable, then the memory dump view will jump to display the data at the desired location.

Modifying data

Analyser allows the user to modify data in the address space of the current process. Double-tapping on a line of data in the memory dump view will display the "Modify Data" dialog box.



The original data is displayed in the current viewing format as well as in hexadecimal notation. The user can enter new data in the same format using the software keyboard. The "Evaluate" button checks the validity of the new data, and translates it into hex. The "Modify" button attempts to write the new data into memory over the old data.

Data cannot be modified while it is being viewed as disassembled SH3 mnemonics. Code must be manually assembled into hex machine code, and entered into memory using the hex viewing format.

When entering hex data, there must be an even number of hex digits in the new data. Single digits cannot be translated into binary data.

PlayTime user guide

Purpose

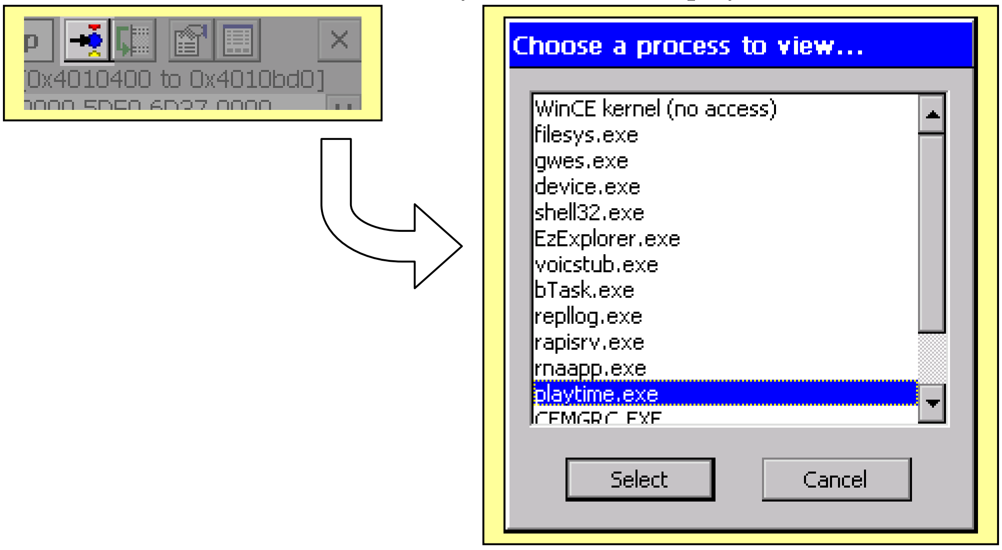
PlayTime is a simple "test bed" application, designed to be modified while in memory. In conjunction with a debugger or memory dump program such as Analyser, the student can learn about the structure of data and its layout in memory. PlayTime has been designed with a simple set of exercises in mind. Various types of data can be modified by the user, and the effects of these modifications can be viewed either within PlayTime itself or by using Analyser.

Using Anlayser with PlayTime!

1. Start PlayTime using the Start menu.

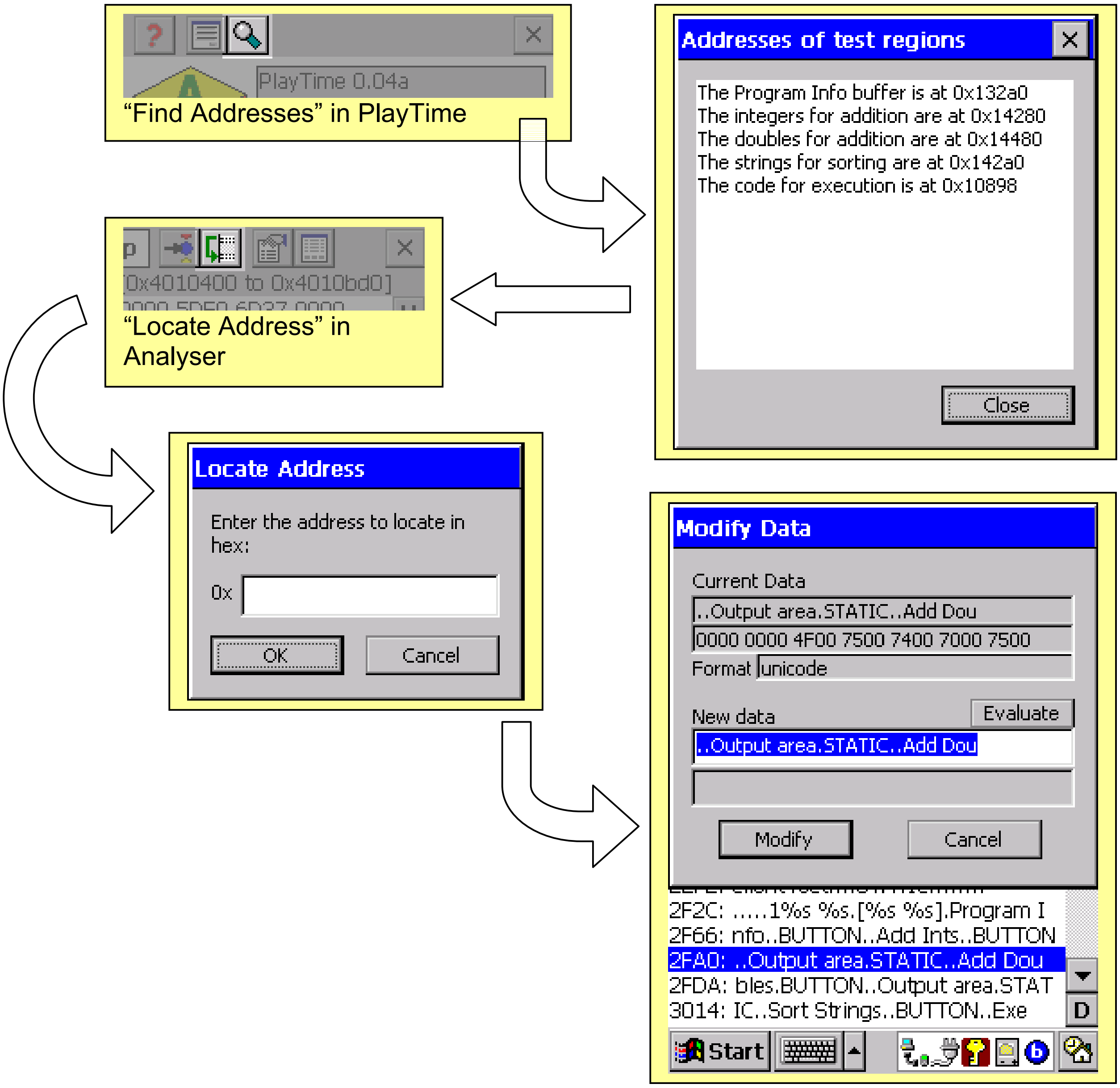
2. Start Analyser using the Start menu.

3. Tap the “Select Process” buttonwithin Analyser, then select “playtime.exe”.



4. Use the View menu within Analyser to change to different viewing formats. The first segment of the process memory space is taken up by the process' code. Switch to Unicode format and use the "Buffer Up/Down" buttons to search for text data and string variables. Use Hex format to compare Unicode data, ASCII data and their hexadecimal values. Keep in mind that the SH3 processor (which is driving the Windows CE box) uses little-endian storage.

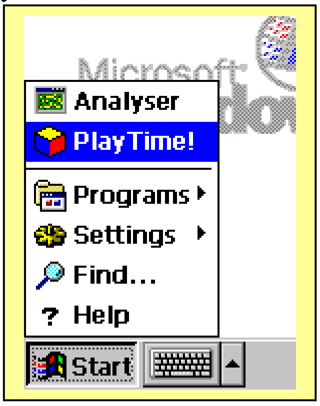
5. Use the "Find Addresses" button within PlayTime and the "Locate Address" button within Analyser to examine the data associated with PlayTime's exercises. Modify the different data regions using Analyser, and examine the results in PlayTime.



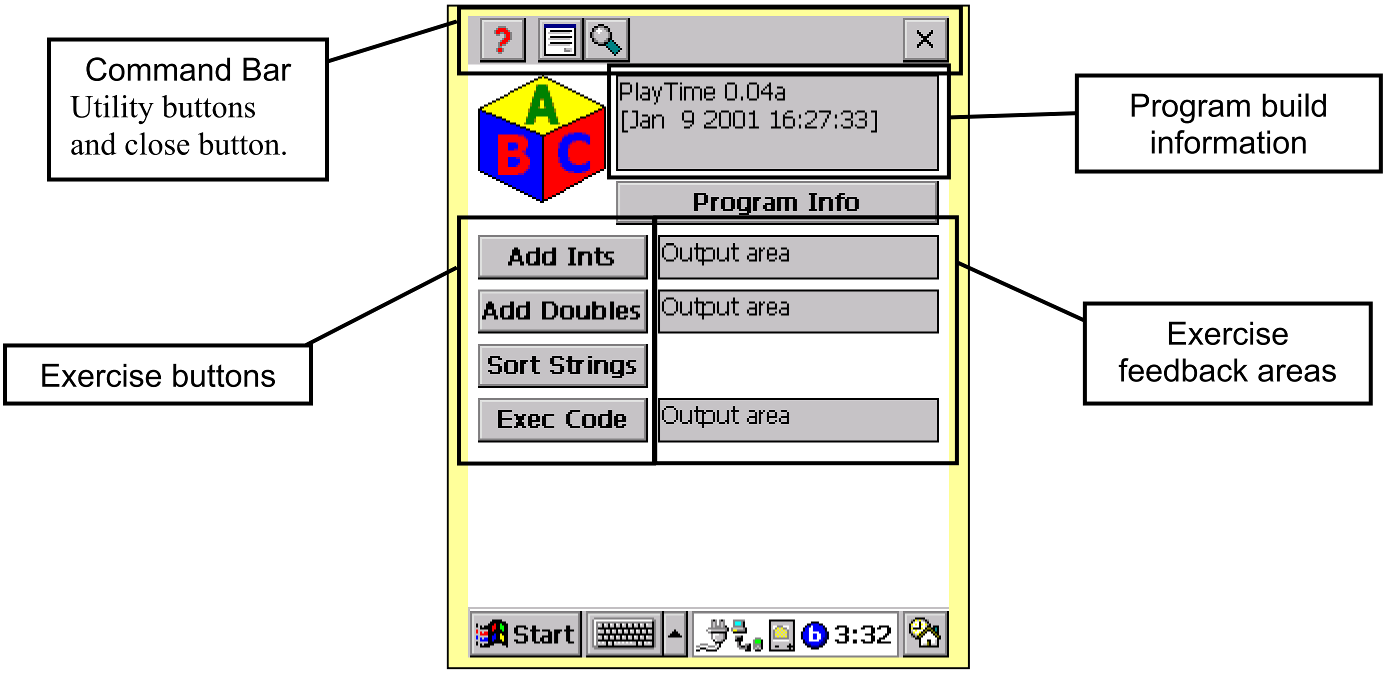
6. Use PlayTime to learn how the different formats of data work. Manually compile some SH3 code.

Main window and command bar

To start PlayTime, use the Start menu.

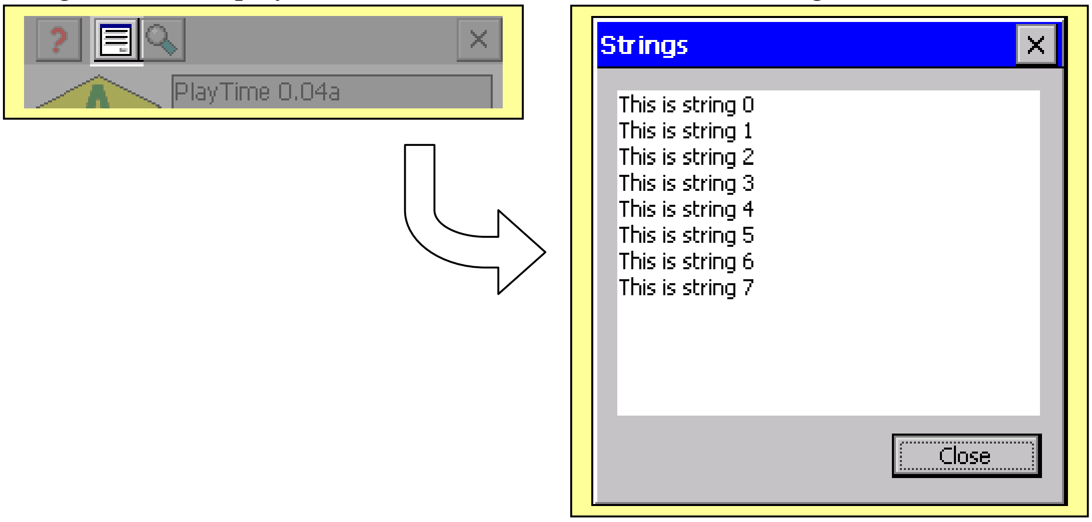


When PlayTime is loaded, the main window appears on the screen.

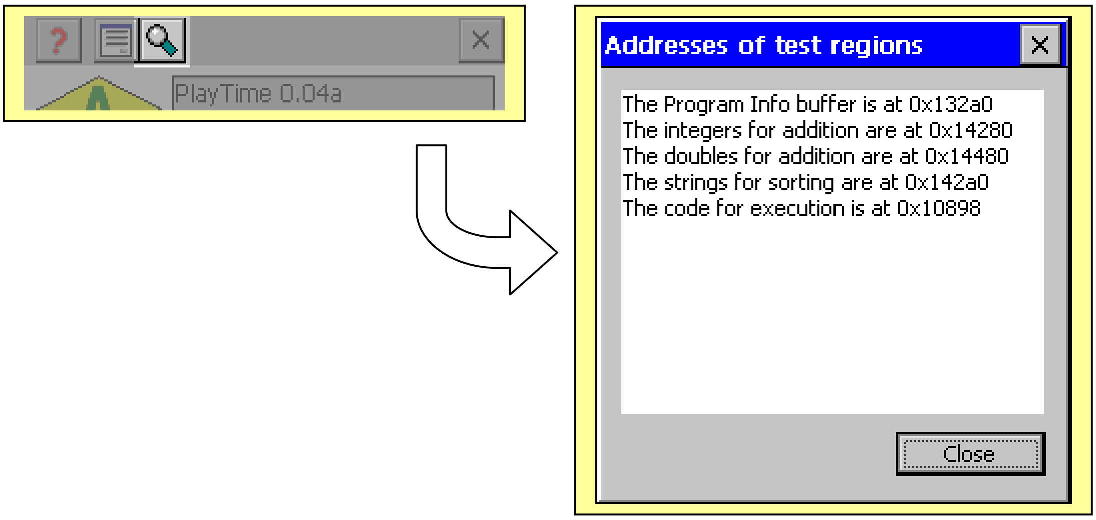


The help button provides a brief overview of PlayTime.

The "show strings" button displays the content of the user-modifiable strings.



The "find addresses" button assists in locating the user-modifiable areas in memory by displaying the addresses of these regions.

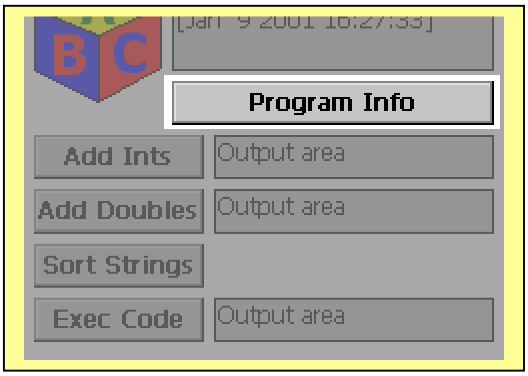


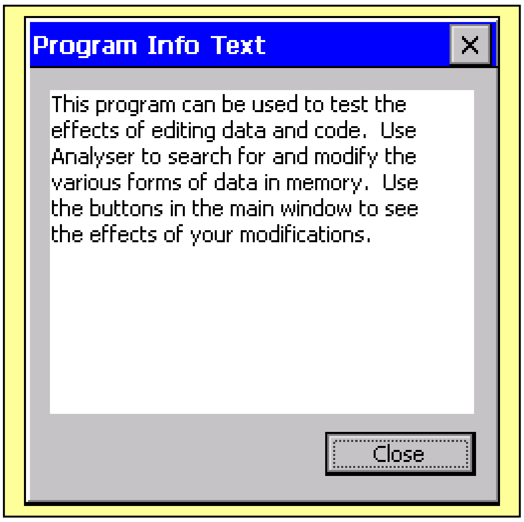
Quit PlayTime by tapping the "close" button.

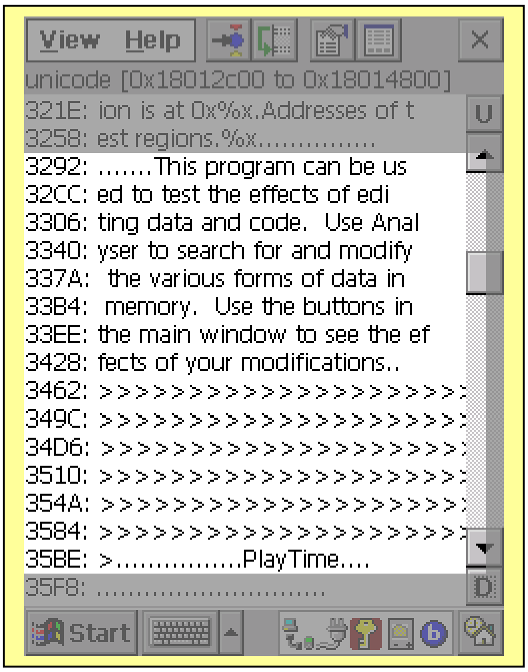
Description of exercises

Program info buffer

The program info buffer is a segment of memory initialised to a paragraph of text concerning PlayTime. It can be modified by using Analyser, and dumped to the screen by tapping the "Program Info" button..







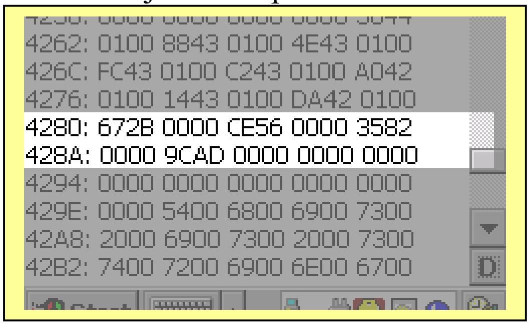
The image on the right is what the program info buffer looks like in memory, viewed with Analyser. The buffer is initialised with the information text, and then padded out with '>' characters.

This is the code used to initialise the data in the C programming language:

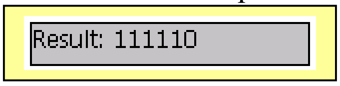
```
#define     MAX_STRING      400
TCHAR       szProgInfo[MAX_STRING];
UINT        uIndex;

for (uIndex = 0; uIndex < MAX_STRING; uIndex++) {
        szProgInfo[uIndex] = L'>';
}
LoadString(hInstApp, IDS_PROGINFO, szProgInfo, MAX_STRING);
```

Integer array

PlayTime provides an array of four integers which can be modified and then summed together by tapping the “Add Ints’ button. The resulting sum will appear in the adjacent output area.



The image on the right is how the integer array is represented in memory, viewed with Analyser. The array is initialised to a sequence of integers summing to 111110_{10}.



This is the code used to initialise the data in the C programming language:

```
int       nAddInts[4];
UINT      uIndex;

for (uIndex = 0; uIndex < 4; uIndex ++) {
     nAddInts[uIndex] = 11111 * (int) (uIndex + 1);
}
```

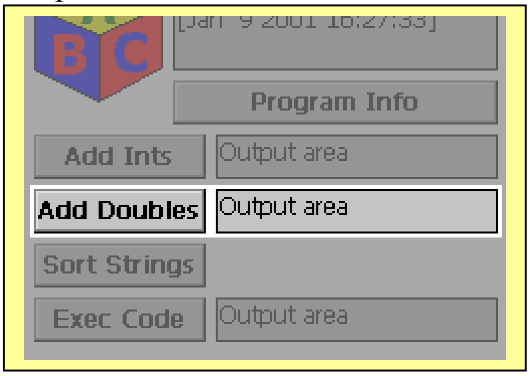
`nAddInts[]`

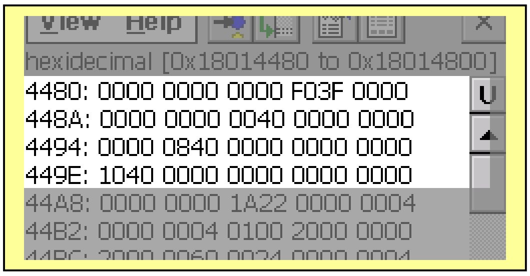
0	11111_{10}	$0x00002b67_{16}$
1	22222_{10}	$0x000056ce_{16}$
2	33333_{10}	$0x00008235_{16}$
3	44444_{10}	$0x0000ad9c_{16}$
Sum	111110_{10}	$0x0001b206_{16}$

The data is stored in memory using little-endian addressing. This means that when a word or longer data structure is placed in memory, the least significant byte (LSB) is stored at the lowest address, followed by the next significant in order up to the most significant byte (MSB) at the highest address. In Analyser, the data appears left to right from LSB to MSB. This is intuitive for single byte data, but reads backwards to how we would write a four digit hex word or an eight digit hex double-word. The double-word `0x12345678`$_{16}$ would appear as “`7856 3412`” in Analyser.

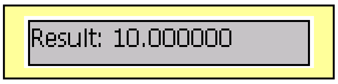
Double array

PlayTime provides an array of four double precision floating point numbers which can be modified and then summed together by tapping the “Add Doubles” button. The resulting sum will appear in the adjacent output area.





The image on the right is how the array of doubles is represented in memory, viewed with Analyser. This array is likewise initialised to a sequence of floating point numbers, summing to 10.00.



This is the code used to initialise the data in the C programming language:

```
double     dAddFloats[4];
UINT       uIndex;

for (uIndex = 0; uIndex < 4; uIndex++) {
     dAddFloats[uIndex] = (double) (uIndex + 1);
}
```

dAddFloats[]

0	1.00_{10}	$0x3ff0000000000000_{16}$
1	2.00_{10}	$0x4000000000000000_{16}$
2	3.00_{10}	$0x4080000000000000_{16}$
3	4.00_{10}	$0x4010000000000000_{16}$
Sum	10.00_{10}	$0x4024000000000000_{16}$

Floating point numbers in Windows CE are stored in the IEEE floating point format. The number is split into three components: the sign (+ / -), the mantissa (the sequence of digits that comprise the number, ignoring the decimal point), and the exponent. The number is reconstructed using the formula

$$num = -1^{S} \cdot (1.M_2) \cdot 2^{(E-127_{10})}$$

In 32-bit IEEE format, 1 bit is allocated as the sign bit, the next 8 bits are allocated as the exponent field, and the other 23 bits contain the normalised mantissa. When converting a number into floating point binary, the digit values are as follows:

…	8	4	2	1	•	1/2 (0.5)	1/4 (0.25)	1/8 (0.125)	1/16 (0.0625)	…

As an example, the steps to convert 9 $97/128_{10}$ (9.7578125_{10}) into IEEE floating point are shown below.

1. Convert 9.7578125_{10} to binary floating point notation.
   $9_{10} = 8 + 1 \rightarrow 1001_2$
   $0.7578125_{10} = 0.5\ (1/2) + 0.25\ (1/4) + 0.0078125\ (1/128) \rightarrow 0.1100001_2$
   $\therefore 9\ 97/128_{10} = 1001.1100001_2$
2. The exponent is currently 0. However, we have to shift the number into 1.xxxxxx… format.
   $1001.1100001_2 \rightarrow 1.0011100001_2 * 2^3$
   The exponent is therefore 3_{10}. The exponent field has a 127_{10} offset.
   $\therefore E = 3_{10} + 127_{10} = 130_{10} \rightarrow 01000110_2$
3. A leading 1 before the decimal point is assumed for the mantissa, so
   $M = 00011100001000000000000_2$
4. The number is positive, so the sign bit is 0.
   S = 0

The final representation of the number is

Sign	Exponent	Mantissa
0	01000110	00011100001000000000000
Bit 31	30 23	22 0

which gives the double-word $0x230e1000_{16}$.

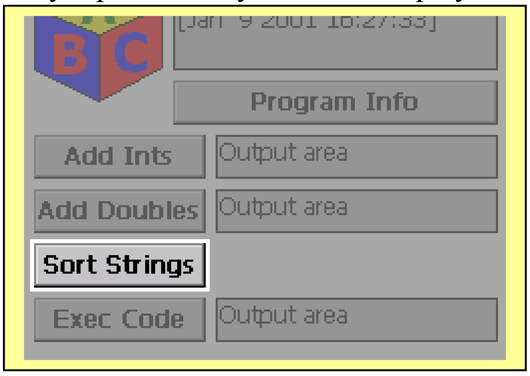
There are some special representations for exceptional cases

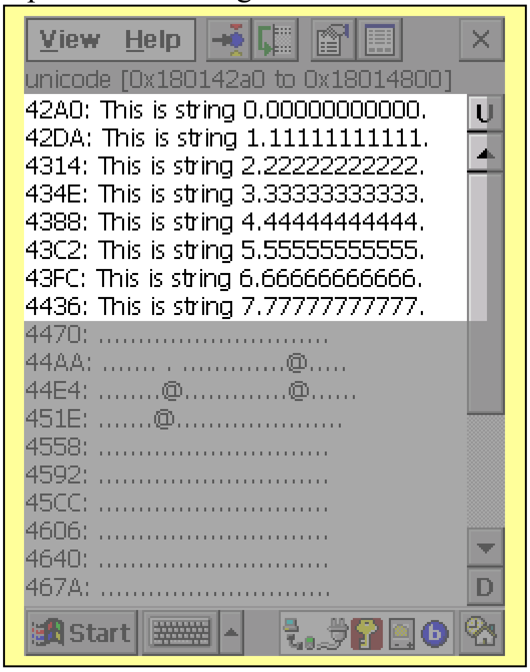
Sign	Exponent	Mantissa	
?	0xff	0x00	+ / - Infinity
?	0xff	NOT 0x00	NaN (Not A Number), Overflow, Error, etc.
?	0x00	0x00	Zero
?	0x00	NOT 0x00	Smaller than smallest precision

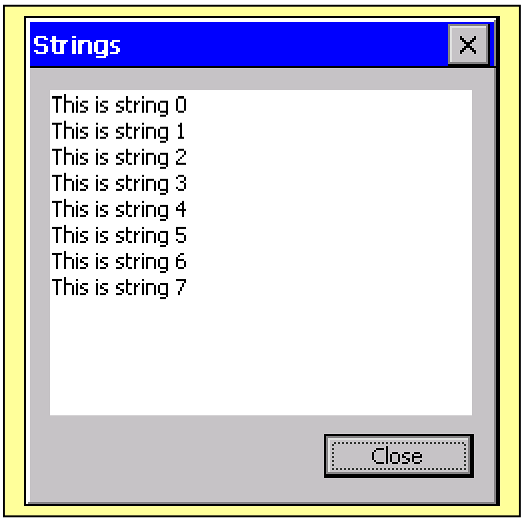
Strings array

PlayTime provides an array of eight strings that can be modified and then sorted in memory. To display the current contents of the string array, tap the "Display Strings" button on the command bar.

To sort the array alphabetically and then display the result, tap the "Sort Strings" button.







The image on the right is how the array of strings looks in Analyser. Each string is padded out with the number of its position in the array. When the "Sort Strings" button is tapped, these strings are moved around in memory to place them in alphabetically sorted order. The sorting performed by PlayTime is case insensitive.

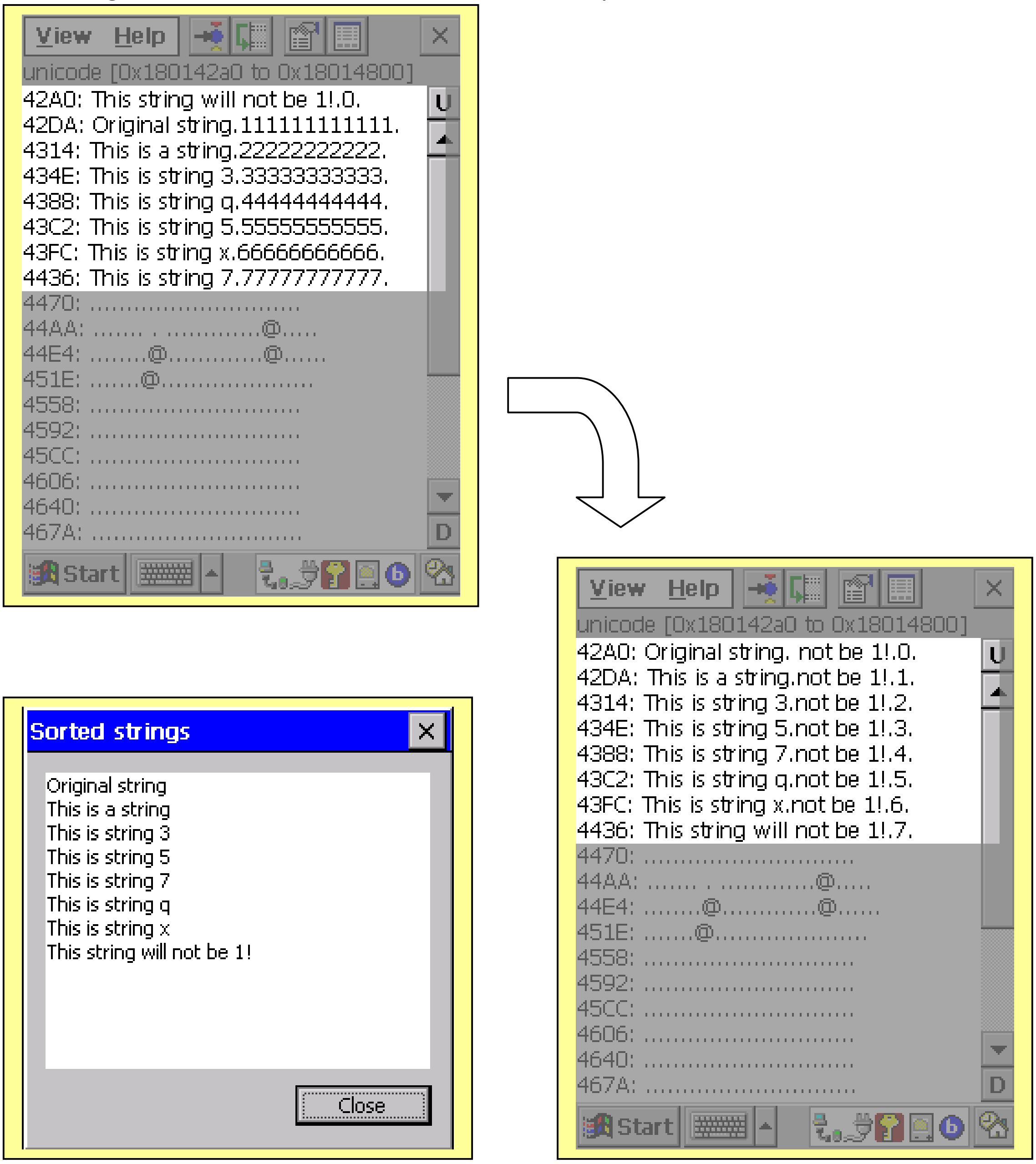
This is the code used to initialise the data in the C programming language:

```
#define    SORT_STRING_LENGTH  29

TCHAR      szSortStrings[SORT_STRING_LENGTH * 8];
TCHAR      *szSort0,
           *szSort1,
           *szSort2,
           *szSort3,
           *szSort4,
           *szSort5,
           *szSort6,
           *szSort7;

szSort0 = szSortStrings;
szSort1 = szSort0 + SORT_STRING_LENGTH;
szSort2 = szSort1 + SORT_STRING_LENGTH;
szSort3 = szSort2 + SORT_STRING_LENGTH;
szSort4 = szSort3 + SORT_STRING_LENGTH;
szSort5 = szSort4 + SORT_STRING_LENGTH;
szSort6 = szSort5 + SORT_STRING_LENGTH;
szSort7 = szSort6 + SORT_STRING_LENGTH;

for (uIndex = 0; uIndex < SORT_STRING_LENGTH - 1; uIndex++) {
     szSort0[uIndex]  = L'0';
}
wsprintf(szSort0, L"This is string 0");

...
```

szSortStrings[]

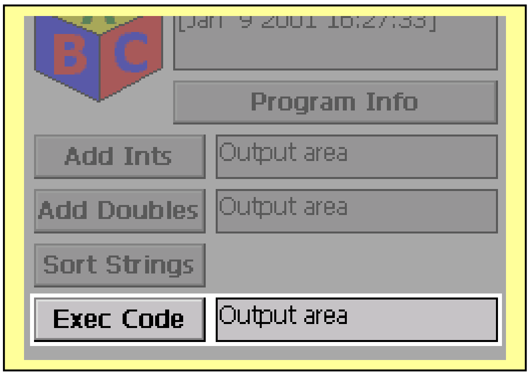
0	"This is string 0"
1	"This is string 1"
2	"This is string 2"
3	"This is string 3"
4	"This is string 4"
5	"This is string 5"
6	"This is string 6"
7	"This is string 7"

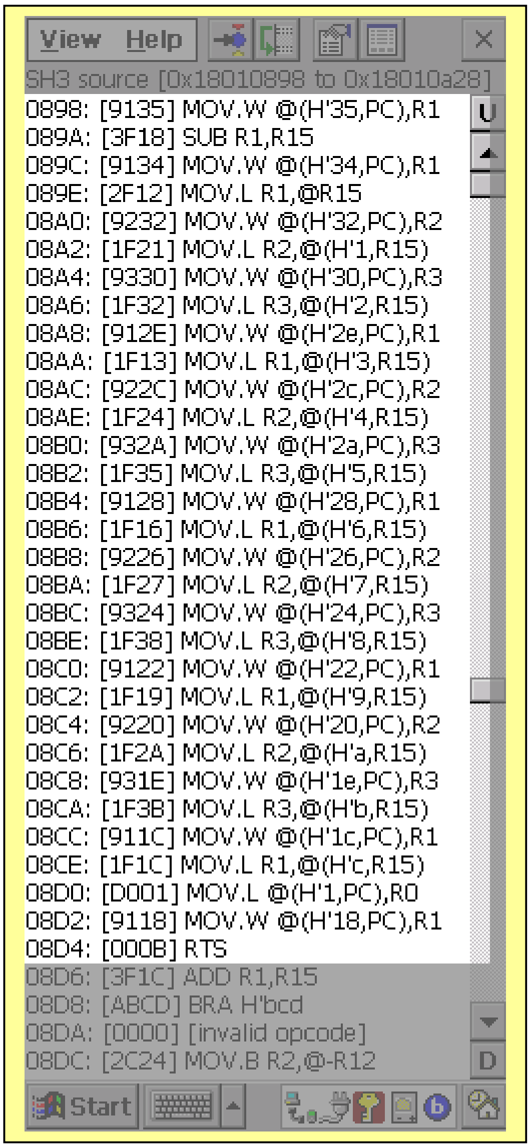
In the example below, the strings were modified using Analyser. When the "Sort Strings" button was tapped, the strings were moved into sorted order in memory.



Modifyable code block

PlayTime provides a segment of its code space for modifying. Using Analyser, the user can manually compile SH3 assembler mnemonics and enter them into the code space in hexadecimal format. Tapping the “Exec Code” button then executes this code and displays the return value.

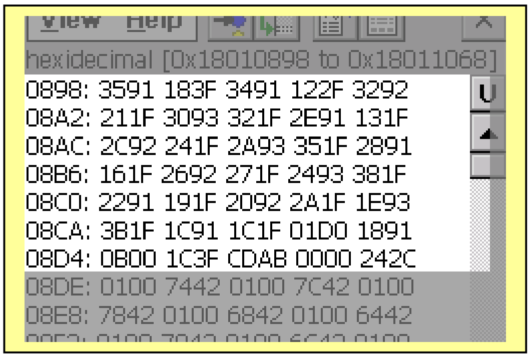




The modifiable region in memory is initialised to a C function which fills an array and returns a hexadecimal value. The image on the right shows the disassembled C function, viewed with Analyser. This region of memory can be modified by Analyser in hex viewing mode, and the function can be replaced with manually assembled user code.

Function source code in C:

```
DWORD ExExecuteCode(void)
{
      DWORD     dwBlock[40];

      dwBlock[0]  = 0x1234;
      dwBlock[1]  = 0x1234;
      dwBlock[2]  = 0x1234;
      dwBlock[3]  = 0x1234;
      dwBlock[4]  = 0x1234;
      dwBlock[5]  = 0x1234;
      dwBlock[6]  = 0x1234;
      dwBlock[7]  = 0x1234;
      dwBlock[8]  = 0x1234;
      dwBlock[9]  = 0x1234;
      dwBlock[10]  = 0x1234;
      dwBlock[11]  = 0x1234;
      dwBlock[12]  = 0x1234;

      return (0xabcd);
}
```



The image on the right shows the assembled code.

Analyser and PlayTime user guides last page